ACORD – Construction and Interrogation of Knowledge Bases Using Natural Language and Graphics – Technical Documentation

Gabriel G. Bès

ACORD – Construction and Interrogation of Knowledge Bases Using Natural Language and Graphics – Technical Documentation

Gabriel G. Bès (ed.)
Université Blaise-Pascal, Clermont-Ferrand

ACORD Esprit project 393, Laboratoires de Marcoussis, December 8 1989

Abstract

The Technical Documentation of the ACORD project resumes five years of work by eight European teams, industrial and academic partners, which involved more than fifty people working around the construction and interrogation of knowledge bases using natural language and graphics.

The Introduction guides the reader towards the architecture of the whole system and its components. The 12 chapters include four common points: scientific background and development, technical description, relation to other ACORD components, potential for development.

See also

Gabriel G. Bès, Thierry Guillotin (eds), *A Natural Language and Graphics Interface: Results and Perspectives from the ACORD Project*, Springer, 1992.

ACORD

Construction and Interrogation of Knowledge
Bases Using Natural Language and Graphics
(Esprit project 393)

TECHNICAL DOCUMENTATION

December 8 1989

Laboratoires de Marcoussis

This document has been edited by G.G. Bès at the Université Blaise Pascal (Clermont-Ferrand II), with full participation from K. Baschung, C. Costes, F. Jurie, F. Lebourg and R. Panckhurst.

Contents

A Reader's Guide

Gabriel G. Bès
CLF

The Technical Documentation of the ACORD project resumes five years of work by eight European teams, industrial and academic partners, which involved more than fifty people working around the construction and interrogation of knowledge bases using natural language text and graphics.

Technical skills alone would not have been enough to obtain final results. A collaborative way of working including workshops, personal contacts, frequent interteam meetings, efficient means of communication and a constant stream of brief internal reports and discussions, made analysing problems, evaluating risks, considering the advantages and disadvantages of a given solution for each partner and for the whole project, and finally decision making, all possible in a coherent manner.

The introduction guides the reader towards the architecture of the whole system and its components.

All chapters include the following common points :

- X.1 Scientific Background and Development
  . . .
- X.2 Technical Description
  . . .
- X.3 Relations to Other ACORD Components
  . . .
- X.4 Potential for Development
  . . .

the art in the domain, which the ACORD project contributed to on many crucial points.

Section X.2 presents a useful technical guide for the understanding and manipulation of software.

Section X.3 defines interfaces with other components and Section X.4 indicates possible developments.

Numbers in bold face (eg. **4**) refer to chapters in this report.

Introduction

Thierry Guillotin
LdM

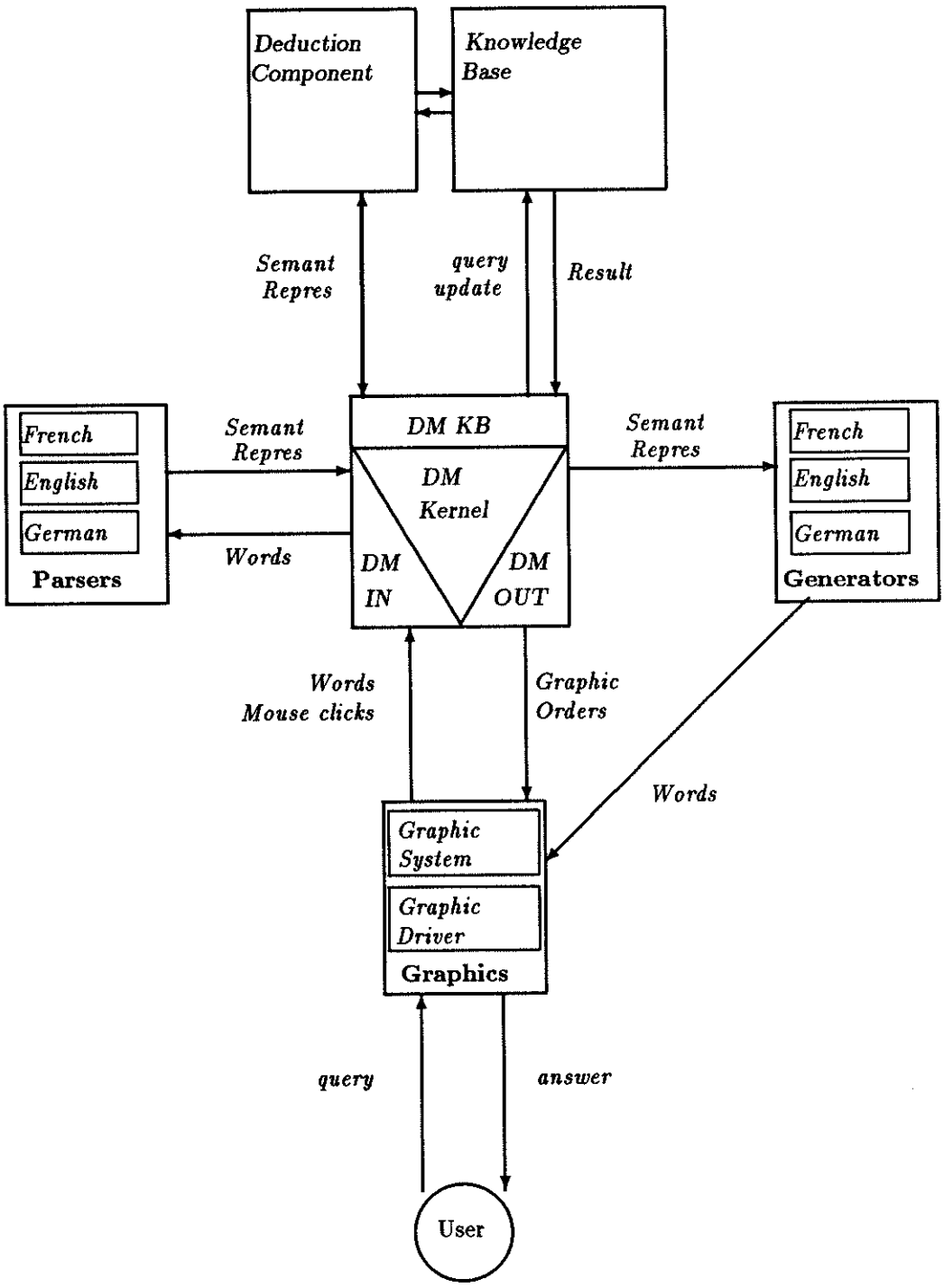
This section presents the Architecture of the ACORD demonstrator and introduces the whole system. The ACORD demonstrator, which integrates all the components of the project, is the coherent result of the theoretical and technical work performed in the project. It must be emphasized that several components, each of which can be demonstrated separately, highlight certain capabilities when represented individually that do not appear in such a straightforward manner when they appear as part of the whole system.

0.1 General Overview of the System Component

The aim of the ACORD prototype is to provide new insights into the use of computational linguistics, semantic and graphic manipulations to build sophisticated Human-Machine Interfaces. ACORD combines Natural Language and Graphic techniques for the creation and interrogation of knowledge bases in French, German and English. The selected application domain is a transportation Knowledge Base (see **11.1**).

The major components of the system, represented in figure 0.1, are the following :

- Parsers for the three languages
- A Graphics component
- A Knowledge Base and a Theorem Prover
- Text generators for the three languages
- A Dialogue Manager including a Resolver and a Planner



DM : Dialogue Manager

Figure 0.1: ACORD System Architecture

0.1.1 The Parsers

There are sub-modules for each input language. Two different syntactic frameworks and parsing tools have been investigated, though there is a common commitment to the general methodology of unification grammar (see **1**). The English and French grammars are designed using UCG (Unification Categorial Grammars) (see **3** and **4** respectively). The German grammar is designed using LFG (Lexical Functional Grammar) (see **5**). In **6**, the Protolexicon, a prototype implementation of a general lexical processor is presented. The three parsers deliver their output in a common semantic language, thus rendering the multilingual aspect for the system transparent to the other components. The semantic language, called InL (INdexed Language), has been developed for ACORD as a derivative from DRT (Discourse Representation Theory).

0.1.2 The Graphics Component

The aim of the Graphics System (GS) is to have the user mix graphics and text in a flexible way and have the system understand the semantics of such mixed expressions. Moreover, the system can also combine graphics and text when displaying results. Such an integration of graphics and Natural Language requires a common means of communication between the two components. The language we use is InL and its derivatives which are produced both by the parsers and the GS and analysed both by the generators and the GS. The functionalities of the GS are :

- deixis pointing : the graphic component displays a map of the current state of the KB. The user can refer to any object on this map by using deictics such as *this, that, here* and *there* in a sentence and specify the referred object by clicking on it with the mouse.
- highlighting : the graphic component can also collaborate to query answering by highlighting an object on the map.
- direct manipulation : the user can move objects on the map and also update the values of the charts associated to truck loads or city depots.

The Graphics System is fully described in **9**.

0.1.3 The Knowledge Base and the Theorem Prover

The Theorem Prover (TP) and the Knowledge Base (KB) are responsible for storing and retrieving the information entered by the user. The Theorem Prover stands as a filter to the KB, retaining general assertions, providing help with complex queries and also formating the input to the KB (see **10**). The KB receives as input DRS syntactic variant to InL formulae and records the information using DRS as Knowledge Representation. It performs a consistency check at each assertion received to satisfy the coherence of

the knowledge stored. To answer the user's queries, the KB uses a Semantic Network to retrieve generic knowledge and modify the initial DRS form query enabling the inference process from queries to known facts (see **11**).

0.1.4 The Generators

The project aims at providing short answers in French, German or English depending on the language used in the query by the user. The first version of the Generator is focused on generation viewed as inverse parsing. The second version is based on a planning component which enables the construction of appropriate referential terms for objects found as the result of the query, i.e. choose between a pronoun, a definite description or a (maximally informative) indefinite description. A full description of the general architecture of generation can be found in **2**. For specific details on English generation, see **3.2.2**; on French generation see **4.2.6**; on German generation see **5.1.5**, **5.2.8** and **5.2.9**. On the PLANNER see **0.1.5** below.

0.1.5 The Dialogue Manager

The Dialogue Manager (DM) supervises the whole system (see **8**). It controls communications between the other subcomponents of ACORD. The interfaces can be briefly described in the following table :

Modules	Input from the DM	Output to the DM
Parsers	list of words	InL with resolution information
Generators	SynInl	list of words
Graphics System	DRS updates	DRS for KB updates
Theorem Prover	DRS	results from KB retrieval

Besides scheduling the transmission of information from one component to the other and processing the output of one component to fit the requirements of the following components, it is also in charge of general purpose processing such as :

- the process called the Resolver which handles anaphora resolutions for the three languages. The Resolver receives InL formulas with resolution information in order to find coherent antecedents and update the dialogue history. The principles of the Resolver are described in **7**.
- the process called the PLANNER which constructs the semantics to be input to the generators. It is written in MACROSSYNINL PROLOG using the InL corresponding to the user query, the dialogue history for appropriate pronominalization and the KB answer. It is described in **2**.

0.2 Technical Information

All these components have been implemented in C-Prolog and in the C language for some of the graphical modules and the ACORD RPC (Remote Process Control) code. The main goal of the architectural design was that all of these modules should communicate quickly and efficiently. The first version of the architecture used UNIX Pipes to implement the link between processes. In 1987 a RPC version was implemented (see 8.1.4 and 8.3.2) which allows to distribute the processes on different machines of a network and provides a better debugging environment.

Within the RPC architecture, there are 4 processes :

- the Graphics System process
- the Natural Language process where the 3 parsers and the 3 generators reside.
- the Dialogue Manager process
- the Theorem Prover and Knowledge Base process

The interaction with the user is made via several windows :

- a graphical interaction window which presents a schematic map of Europe showing a part of the current KB state
- a textual interaction window where the user can type his/her queries and assertions and also receive ouput from the system
- moreover, for each of the processes described above, a diagnostic window is provided where main events are traced.

Resumed content of directories and files The modules are structured in the main directory acord as follows :

Makefile	general Unix Makefile for all directories
epi	English Component
ldm	French Component
stut	German component + the generation planning part
Resolver	Anaphora Resolution Component
edc	Graphics Component
fhg	Dialogue Manager
rpc	Remote Process Control Component
ta	Knowledge base
bull	Theorem Prover

Chapter 1

The Grammatical Environment of UCG Grammars

Jonathan Calder,
Neil Leslie,
Mike Reape
ECCS

The PImPLE system for grammar development and testing is described in this chapter. The module also defines the syntax and semantics of UCG objects. PImPLE has been used within the ACORD system for the development of the French and English grammars (see 3 and 4). We describe first the scientific background and history of the component and then discuss its implementation and use.

1.1 Scientific Background and Development

General design The PImPLE system allows a user to write a grammar and test and debug it against sentences typed into the computer. To aid in this work a number of facilities are provided which allow the system to take over some of the work involved in the organization of a grammar. These facilities are also necessary because of the overall design of the system which aims to favour compile-time processing over run-time processing.

The system we describe here has benefited from several years' experience in the implementation of unification grammars and grammar development. The key references cover the development of Functional Unification Grammar (Kay 79) and of PATR-II (Shieber et al. 83, Shieber 86) including its descendants such as D-PATR (Karttunen

86). The recent literature on formal properties of unification, with special reference to linguistic applications, includes Pereira and Shieber 84, Johnson 87 and Kasper 87. In the following discussion, we will be somewhat loose with our terminology, using the terms *feature*, *attribute* and *label* interchangeably. We will also follow PROLOG syntactic conventions.

PImPLE's original design was as an interpreter for PATR-II-style grammars, including most of the features described in Shieber et al. 83. These include the definition of *templates* to capture patterns of recurring information, *lexical rules* to state relations between feature structures, *path abbreviations* to abbreviate common paths in feature-value structures and *grammar rules* to state how smaller expressions combine to form larger ones. This implementation used the algorithm by Eisele and Dörre 86 for the unification of directed acyclic graphs (DAGs).

It was soon realized that the generality of the DAG formalism was not only greater than that required by UCG, but also prevented the direct use of PROLOG's unification mechanism for combining the information expressed in grammatical objects. To sidestep this problem, it was decided to require the user to make statements about the structure of grammatical objects, in the form of typing declarations. The procedure for interpreting a grammatical object checks that it is well-formed with respect to those declarations and as a side-effect constructs a PROLOG term.

The type-checking algorithm used by the system is based on Cardelli 84. This approach allows single-pass compilation to terms, in contrast with much others in the implementation of unification-based languages in PROLOG where an initial pass is required to determine the set of labels used in a grammar, prior to the construction of the PROLOG representation of a particular object.

The use of disjunction and negation in unification formalisms was introduced by Karttunen 84. Disjunctive specifications are useful both for conflating grossly similar feature structures and for reducing the number of partial analyses stored in the course of parsing a sentence. Negative specifications allow the concise statement of disjunctive constraints. Because of doubts about the soundness of unification algorithms involving disjunction and negation at the time of implementation, we restricted ourselves to *atomic* or *value* disjunction and negation. That is, one may state that a particular value may be one of a set of particular constants (in our case PROLOG atoms) or may not have a particular constant as its value. One may not state that two feature structures are distinct, or that a particular feature has one of a set of feature structures as its value. These restrictions allow PROLOG variable instantiation and unifiability as tests for the evaluation of a constraint.

The French grammar used in the ACORD system has the additional syntactic construct of *correlational constraints*, in which one may conditionalize the value of some feature on the basis of the value of some other feature in the feature structure. That is, one may say that if a feature F has some atomic value V, then some other feature F' has some (possibly atomic) feature structure as its value. If the condition doesn't hold, some other description may be associated with the feature structure in question. The

syntax of these constraints is discussed in more detail in **4.2.3** below.

The current implementation of PImPLE consists of a number of sub-components with the following purposes.

- The UCG reader and printer

  The module implements the functionality of PATR-II, extended with disjunction and negation over atomic values. This permits the definition of *templates*, *lexical rules*, *path abbreviations*, *lexical items* and *grammar rules*. In addition, this module defines the syntax and semantics of the representation of UCG used in Deliverables T1.6 and T2.6. The basic strategy has been to allow a form of representation closely related to published work on UCG, for reasons of conciseness and comprehensibility. We use the technique of using the stated typing properties of objects to allow a simple, one-pass compilation of statements into PROLOG terms.

- The sort system

  It is convenient to allow the definition of complex constraints which feature structures must obey. This module allows the definition of such complex constraints in terms of the propositional calculus. These constraints may be attached to variables and constants in the representation of UCG objects. This system is described in Deliverable T1.6.

- Protolexicon definitions and entries

  This module is discussed in Deliverable T1.12 and see **6** below.

- A shift-reduce parser and a head-driven generator

  The system contains an extremely simple bottom-up tabular shift-reduce parser, described in Deliverable T2.6. It has been specialised on the basis of known properties of the grammar required by the ACORD system, in particular that in any situation a finite number of unary rules may apply to a particular constituent. The generator is described in Deliverable T2.10 and see **2** below.

- Menu-driven interaction

  Interaction with the system is conducted primarily through the use of menus. These allow the user to test sentences, examine the results of parsing and selectively recompile parts of the grammar.

1.2 Technical Description

1.2.1 Software and Hardware Requirements

PImPLE is implemented in PROLOG, and runs under various dialects of that language. The storage requirements of the system are as follows, including the Protolexicon (these figures are approximate and are given for `prolog+`):

Disk storage	500 kbytes
Run-time heap	256 kbytes with no grammar loaded
Run-time heap	1400 kbytes with the English grammar loaded

For PROLOGs which cannot allocate storage dynamically, further allowance has to be made for the storage of atoms and for heap required at run-time for the storage of partial results. As described below, the design of the parser requires a large amount of memory.

Files Source code for PImPLE and the Protolexicon (see 6) are to be found in `acord/epi/Protolexicon/src`. Files with the extension `.pl` are PROLOG source files. Files with the extension `0.pl` contain code which must be installed with certain site-dependent information before use. A file called `MANIFEST` in this directory gives more details of the contents of the individual files.

1.2.2 Software Description: General Design

Implementation strategy In the current implementation, we make as much use as possible of the facilities of C-Prolog's "secondary database". This database provides fast access to stored information by means of a key and gives a means of emulating the indexing performed by PROLOG's such as Quintus[1] Prolog. PImPLE objects are typically stored in the secondary database under a key which represents their type, using the following structure:

```
object(Object, Key, File)
```

where `Object` is the result of compilation, `Key` is the key under which a particular item is stored (its occurrence here is strictly redundant), and `File` is the name of the file in which the object was defined. For some objects, such as lexical items and grammar rules, an alternative representation is also required. In order to obtain constant time look-up for lexical items, a different scheme is used, described in detail below. Briefly, each lexical item is stored under a key generated by the concatenation of the symbol `#` with the string form of the item in question. All other keys are represented as the concatenation of the symbol `&` and the key name.

The following types of PImPLE object are recognized (ignoring those required by the Protolexicon):

[1] Quintus is a trademark of Quintus Computer Systems, Inc.

A	`template lexical rule rule path abbreviation declare`	
B	`properties`	The set of propositional variables used by the sort system
	`property_term`	Internal representation of the translation of a property
	`axiom`	The user-definition of an axiom
	`axioms`	The set of all axioms
	`user_connective`	A user-defined logical connective
	`sort_definition`	The definition of a sort
	`sortdefprops`	The most specific conjunctive definition of a sort (used in printing routines)
	`dag`	Field selectors for the construction of terms from feature descriptions
C	`dm_test`	User statements on sortal restriction of relations.

Those in group A correspond to the definitions of templates, lexical rules, path abbreviations and typing statements used by the UCG reader. Those in group B are used by the sort system compilation routines. `dm_test` statements represent sortal restrictions over relations and are used in the ACORD grammar primarily in the elimination of prepositional phrase ambiguity.

File handling Each user-supplied file is defined to be of a certain type, these types are:

`axiom`	Axioms of the sort system
`sort`	Sort definitions
`definition`	Path abbreviations, typing declarations, templates and PATR-II-style lexical rules
`lexical_definitions`	Lexical entries
`grammar`	Grammar rules

Each of these types is associated with one or more compilation routines, given by the predicate `in_file/2`, whose first argument is a file type and its second a list of routines, annotated with either `once` or `resatisfiable`, indicating whether it is possible for a compilation to result in more than one object. (This will happen when an object is defined in terms of a disjunctively defined template, for example). An example of `in_file/2` is

```
in_file(definition,
        [declaration:once, path_abbreviation:once,
         u_mk_template:resatisfiable, lexical_rule:resatisfiable]).
```

which corresponds to the third file type given above.

The load sequence for a particular file is defined by the predicate `p_load_file/2` whose arguments are a file name and a file type. First, the routine `remove_from/1` is called with the name of the file to be loaded as its argument. This removes all definitions previously loaded from that file. The statements in the file are then read in turn and

passed to the routine `p_obj/5`, together with the set of valid keys, the type of file being loaded and the names of compilation routines valid for that type. This routine is responsible for invoking the compilation routine and for reporting errors in the case where no compilation can be performed. It returns the result of compilation, together with a key. A compilation routine is called with the item read from the file as its first argument and it returns the result of compilation as its second argument and a suitable key as its third.

The result of compilation, the associated key and the file name are then passed to the routine `p_note_def/3`. A check is also made to determine whether the key is one of the standard keys given above, or a key for a lexical item. In the latter case, a further record must be made, in order to allow the deletion of lexical entries, via `remove_from/1` in the case where the user has edited a file of lexical definitions and wishes to replace their old definitions by their new ones.

The files that define a grammar are loaded in the order given in the above table. File names are determined by the grammar name and an extension appropriate to its type (see the section on *Variables* below). However, the user is not restricted to file names constructed this way. In any file, it is possible to use the form

```
source(FileName).
```

to indicate that `FileName` should also be compiled as a file of the type being loaded.

Objects are retrieved from the PROLOG database by use of the evaluable predicate `recorded/3`, whose first argument is a key appropriate to the type of object required.

The UCG reader and printer The following section defines the syntax and semantics of the parts of the UCG reader and printer. Fuller examples of definitions are given in Deliverable T1.6.

Path-based descriptions Following the syntax of PATR-II, we allow path-based descriptions of feature structures. There are minor difference resulting from our decision to make all objects processed by the system readable by PROLOG's standard reader. A path-based description is then a PROLOG list of descriptions, where a description has the following form:

```
LHS = RHS
```

where LHS is a path, that is a sequence of PROLOG atoms separated by the path constructor `:`. RHS may be an atom, representing a constant value, a template name preceded by `@`, a path or a term-based description preceded by `+` (described below). The following is then a path-based description.

```
[:category:cat = noun,
 :semantics:index:kbclass = :pred,
 :occlist:position = :occnumber].
```

Path abbreviations A statement such as

```
Abbreviation      abbreviates      Path.
```

indicates that whenever `Abbreviation` appears as an element in some path, it is replaced by the `Path` given. An example of this is

```
pred abbreviates semantics:predicate.
```

This abbreviation is used in the second line of the description above.

`declare` Typing statements Typing statements define the legal instantiations of a feature structures and state which feature names may cooccur within it. Put another way, they allow the grammar writer to state those features or attributes which exhaustively describe a particular object. Thus, UCG recognizes eight attributes of a sign in the following definition:

```
declare(sign,
        phonology ** {phonlist, basic}:
        category ** {complex, bcat, basic}:
        semantics ** {inl_variable, semantics, occlist, basic}:
        order:
        in ** {sign, basic}:
        out ** {sign, basic}:
        occlist ** {basic, occlist, improper}:
        occnumber).
```

The first argument to a `declare` statement gives the type of feature structure we are describing. Its other argument is a colon-separated set of feature names or labels, possibly annotated by the typing operator `**` and a set of types. `basic` is the only type that is provided by the system and corresponds to the PROLOG `atomic/1`. If a label has no annotation, the interpretation is that that label may be used in a feature structure of that type provided the corresponding value is of type `basic`. If a label has an annotation, any instantiations to that feature must be compatible with the types given in the annotation. Two feature structures are type-compatible if they have identical sets of labels associated with them.

In addition to the constructor :, the grammar writer may also use arbitrary PROLOG operators to define types (and use them in term-based descriptions). An example of this is the use of / to encode the categorial slash of UCG, as defined by

```
declare(complex, remainder ** {basic, complex, bcat} / active ** sign).
```

which says that a feature structure of type `complex` may have two labels associated with it, `active` and `remainder`.

Each typing statement is represented internally in a form similar to that above, with the addition of a non-ground term that represents the feature structure as a whole and variables associated with each label which act as field selectors in the term. Also a term that represents the most general description of the type is included for the purposes of the term-based description system. So part of the internal representation for the above typing statement is:

```
declare(sign,
        spec(phonology, [phonlist, basic], A):
        spec(category, [complex, bcat, basic], B):
        spec(semantics, [inl_variable, semantics, occlist, basic], C):
        ...
        _:_:_:_:_:_:_:_,
        sign(A, B, C, D, E, F, G, H))
```

For the construction of terms from path-based descriptions, explicit field selector clauses are also defined. An example of this is

```
d(category, sign(A, B, C, D, E, F, G, H), B, [complex, bcat, basic])
```

where the fourth argument is the type of the selected field. These clauses are the basis of the predicate `p_path_value/3`, for the construction of terms from path-based descriptions.

A special typing statement takes the form

```
top_level_type(Type)
```

which gives the type to be used in interpreting user definitions without explicit labels. We call these descriptions *term-based.*

Term-based descriptions A term-based description is one which uses the typing information provided by the grammar writer to interpret the grammar. By removing the requirement that feature structures must have explicit labels, more concise statements are possible. In this case, the user supplies definitions in forms which are similar to those used in typing statements. In particular, a user definition is of a given type if it is structurally identical (i.e. uses the same number of operators and fields) to the type definition.

The following term-based description contains the same information as the path-based description shown above:

```
(_: noun~_: ( (KBClass\_):KBClass:_):_:_:_: (_:Number:_:_:_):Number)
```

Boolean values The value of a feature may be a complex value constructed via the operators ~ for negation and or for disjunction. Such values may be used with either term- or path-based descriptions. The routine which analyses such expressions is called `u_eq_type/2`. This routine creates the internal representation of a disjunction or negation which is then associated with a PROLOG variable. The internal representation is in one of two forms, the first for a disjunctive constraint, the second negative:

```
'C'(Variable, 'OR'(_, ListOfDisjuncts))
'C'(Variable, 'NOT'(_, Atom))
```

The routine `check_constraints/2` is called whenever a unification has been performed, with its first argument the result of appending the constraints associated with the two objects unified. The discussion of unification contains more details on this topic (see section **The definition of unification** below).

Templates Templates come in two forms. These are:

```
TemplateName aliases Description.
TemplateName => Description.
```

where `TemplateName` is a PROLOG atom and `Description` is a term- or path-based description. The first form indicates that the template name may be considered as an abbreviation for the purposes of display routines.

Templates are processed by the routine `u_mk_template/3` and the results of compilation are stored in the form of a triple, `template(Name, Alias, FeatureStructure)`, where `Alias` may have the value `alias` or `not_an_alias`.

Lexical items PImPLE lexical items may be either term- or path-based. In both cases, the string form appears to the left of the operator :. In the first case, the typing conventions indicate that the string form is also the value of the feature phonology. In the second, the string form has to be explicitly inserted as the value of this feature. Lexical rules may also appear in path-based lexical definitions, using the syntax:

```
String: [ ..., LexicalRuleName, ...].
```

(Note that this case is distinguished from that of templates by the absence of the template marker @). The interpretation of such definitions follows PATR-II.

As discussed above, lexical entries are stored in the PROLOG database using a key constructed of the concatenation of # with the string form of the item. Its representation in the database takes the form:

```
dict(String, FeatureStructure).
```

To reduce the storage space required for lexical items, a distinction is made between "simplex" and "complex" definitions. A simplex definition is one which consists purely of templates and lexical rules. In this case, the definition is stored in its textual, rather than its compiled, form. This is possible as the main cost of converting such a definition into its compiled form is the unification of feature structures.

Lexical look up is performed by the routine `p_retrieve_word/4`, called with the string tokenised as a word by the reader. The concatenation of the symbol # with this string is then used as a key for lookup. `p_retrieve_word/4` also takes as argument the vertex number to be associated with the left side of the lexical edge. The user may ask for this number to be inserted into the feature structure (see the section on *Variables* below). Various subsidiary routines handle cases specific to ACORD having to do with proper names consisting of a noun and numerical identifier.

Lexical rules Lexical rules have the syntax:

```
lexical_rule(Name, FeatureStructureIn, FeatureStructureOut).
```

where `FeatureStructureIn` and `FeatureStructureOut` are term-based descriptions. The routine `u_lexical_rule/3` creates the internal representations of the form

```
lexical_rule(Name, lr(In, Out)).
```

where `In` and `Out` are compiled feature structures.

Grammar rules Grammar rules have the syntax

```
ba =>
        Mother --> D1, D2, ..., Dn.
```

where the Mother and daughter feature structures are term-based descriptions. These are compiled into the internal representation

```
rule(Name, MotherFeatureStructure, Daughters)
```

by the routines rule/2. A further step in compilation is discussed in the section describing the parser, below.

The UCG sort system The English PImPLE system contains a facility for adding *sort information* to arbitrary Prolog terms and subterms. This sort information is used in the English grammar to represent the UCG sort system. Sorts can be arbitrary formulas of the propositional logic (in constrast to other types of propositional sort systems). The syntax and use of the sort system is described in ACORD Deliverable T1.6 and so will not be described here. What we will describe here is the file sort.pl and its associated files. These files contain the predicates which compile sort formulas into their internal representation and predicates for manipulating those predicates and for checking the consistency of sort information.

We will begin with sort.pl.

The top level predicate for compiling the sort system is encode_sort_system. It manipulates two data structures: $ModelNumber$(N) which is a counter of the number of models produced so far and bad_axiom which is a flag which simply indicates that compilation of some axiom, sort definition, etc. has failed. encode_sort_system simply removes the flag and clears the counter. It then calls build_encoding to compile all of the sort definitions in accordance with the axioms that define the lattice of sorts. It then clears $ModelNumber$ to 0 and then records the number of models produced as a result of compilation.

build_encoding does the main work of compiling the sort system. Compiling the system consists in determining what set of *models* is admitted by a set of axioms. Recent axiomatisations in the English UCG grammar have yielded roughly 40-100 models although the number of possible models (with no axioms) is of the order of 2^{13}. A model is simply a PROLOG term of the form model(Model) where Model is a list of *literals*. A literal is a PROLOG term of the form +Property or -Property where Property is one of the *properties* declared in the properties declaration. A property corresponds to a propositional variable in propositional logic. Therefore the model model([+human,-female]) is a model where the propositional variable human is assigned the truth value

T and female is assigned the truth value F. The axioms can be looked at as *meaning postulates* which restrict the class of admissible models.

build_encoding begins by collecting all of the properties which have been read in by the PImPLE reader, sorting them by alphabetic order and then storing them in the standard way using p_note_def. All of the properties are processed at one time rather than incrementally as with most other PImPLE objects to allow them to be sorted. This is convenient for making models easier to read and aids in debugging sets of axioms. The set of axioms is then collected and stored in the standard way. They are processed at one time because it is impossible to compute the set of models admitted by a set of axioms incrementally. translate_axioms(Axioms) is then called to translate metasyntactically defined connectives into a set of *basic connectives*. These will be described below. translate(Axioms) is then called to actually produce the models admitted by the axioms. encode_properties(PropertyList) is called to determine which models a property holds in. Finally, encode_PropertyList(PropertyList) is called to translate the list of models that a property holds in into a bit encoding. This is described in more detail below.

translate_axioms takes user defined axioms and translates them into the set of basic connectives. These include at-most-one-of (amo(A)), inclusive-or (or(A)), and (and(A)), exclusive-or (xor(A)), conjunction (&), disjunction (or), negation (-), iff (<->) and if (->). In addition, positive literals may be specified with a +, e.g., +human. Any user defined connectives are also translated into the set of basic connectives here. User defined connectives may be defined in terms of anything which can ultimately be translated into the basic connectives. This includes other properly defined user connectives.

There are also a set of four connectives from *systemic grammar*. Their forms and translations are given in the following table. They have not been used extensively.

```
'-|'(A,List)              (A & xor(List)) or ((-A) & (-or(List)))
'-{'(A,[X1, ... , Xn])   (A <-> X1) & ... & (A <-> Xn)
'}-'(List,A)              A <-> and(List)
']-'(List,A)              A <-> or(List)
```

Any unknown properties or connectives are flagged as errors and the relevant axiom is deleted from the list of axioms. Processing still continues despite errors. This may cause problems if the effect of continuing is to create very large numbers of models. The flag bad_axiom is used to "remember" if there has been an error during the translation of an axiom. By doing this instead of failing immediately, all errors in an axiom are reported.

translate(Axiom) actually does two things. The first half of its functionality consists of half of a disjunctive normal form construction algorithm for the propositional logic. The second half of its functionality calculates a disjunct of the disjunctive normal form of a formula and creates all the model terms which are consistent with it.

We say that `translate` is half of a disjunctive normal form algorithm because it doesn't actually calculate the disjunctive normal form of a formula but instead does the other rewriting normally expected of a normal form for propositional logic. `expand_disjunctions` then calculates the disjuncts in the normal form on backtracking. This is to prevent requiring an exponential amount of space to calculate the formula. The normal form calculation is based rather closely on that presented in Clocksin and Mellish 81. It first calls `conj_from_list` to convert the list of axioms into a conjunction. Then `implications_out` is called to rewrite implication and the biconditional in terms of `&`, `or` and `-`. Next, `negation_in` is called to eliminate negations of all formulas except properties. Next, `cnf` (which stands for `conjunctive_normal_form`) is called to rewrite the formula with respect to various optimisations. Currently this is just defined as the unit clause `cnf(X,X)`. (Originally, it was hoped that this would make the algorithm faster but the overhead involved in the rewriting only appears to slow things down.)

`expand_disjunction` calls `expand_disjunction0` to generate a model by calculating one of the disjuncts of the disjunctive normal form DNF of the formula. The *dnf* of the formula will be a disjunction of conjunction of *literals*. A literal is a term of the form `+Property` or `-Property` where `Property` is a property. Then `expand_model` converts the disjunct into a list of literals. Not every property will occur in a literal so it then nondeterministically adds `+Property` or `-Property` to the list of literals for every property which does not occur in a literal. The result is a list of literals, one for each property. Such a list is a *model*. The model is saved and then the clause failed to investigate other models implied by the disjunctions. Notice also that any one disjunct can give rise to more than one model. This is the only practical solution to eliminating the disjunction. A straightforward term rewriting approach is exponential in time and space. This scheme has the advantage that inconsistent models are eliminated as quickly as possible which would not be the case with a real DNF algorithm.

`expand_disjunction0` calculates a disjunct in the DNF by nondeterministically choosing one of the disjuncts in any disjunction and then recursively invoking itself on that disjunction. The crucial clause is the following.

```
expand_disjunction0((A or B),L0,Ln) :- !,
  ( expand_disjunction0(A,L0,Ln),
    (L0 = Ln, ! ; true) ;
    expand_disjunction0(B,L0,Ln) ).
```

The basic strategy is to collect literals in a list. The second argument to `expand_disjunctions0` is the input list and the third argument is the output list. Previously unseen literals are added to the end of the list. We can call such a list a *partial model*. Thus, the partial models are threaded through successive calls to other predicates in the clauses for `expand_disjunction0`. The predicates `positive/3` and `negative/3` check that a positive or negative literal respectively is consistent with the current model and adds the literal to the model if it is not already there. If the literal is not consistent with the model then it fails causing another disjunction to be examined.

The clause above incorporates a test to prune redundant parts of the search space.

For essentially disjunctive connectives, we can distinguish between formulas which are satisfiable and don't instantiate any further properties (i.e., which are satisfiable on the basis of the current property instantiations in the partial model) and those which do instantiate further properties in the partial model. Those that don't further instantiate a model will obviously return a partial model identical to the input model. In these cases, we can cut the other disjuncts in the disjunction being considered from the search space because the instantiations which make the disjunction true were not effected by the disjunction itself. As long as the relevant instantiations are not inconsistent with any further formulas, the disjunction is guaranteed to hold and any way of satisfying the disjunction will include these instantiations. Therefore, the property instantiations implied by the other disjuncts need not be specified. In the event that the relevant instantiations are inconsistent with further formulas, then the disjunction as a whole will fail anyway precisely because it was not the disjunction which effected the instantiations in the first place. That is, to undo the invalid instantiations will require backtracking past the disjunction. Therefore the clause above tests that the input and output partial models are the same after `expand_disjunction0` is called on the first disjunct and if they are it cuts thus preventing consideration of the second disjunct.

This strategy can be eliminated if any higher level disjunctive connectives are translated into conjunctive normal form CNF. The tradeoffs are very subtle and involve interaction between level of compilation, the PROLOG backtracking strategy, avoidance of a generate and test scheme and the use of partial models. There are three connectives which do not get translated into CNF but are effectively interpreted. They are `xor`, `or` and `and`. In the case of `or` and `and` the negations are just translated into the equivalent negated form using the predicate `negation_in` and then executed. It was decided that the translation approach for `xor` was too expensive and so its negation is interpreted. This necessitates the interpretation of the other two as well in their positive form.

Other connectives which are treated this way and formulas involving them are `or(List)`, `xor(List)` and `-xor(List)`. They are treated by `inclusive_or/3`, `exclusive_or/3` and `neg_exclusive_or` respectively.

Then `encode_properties(PropertyList)` creates a list, `PropertyList` of lists where each of the sublists of `PropertyList` is a list of negative and positive literals of each property. I.e., if the set of properties is `human`, `object` and `plural`, then `PropertyList` might be:

```
[[-human, +human, -human],
 [+object, +object, -object],
 [-plural, +plural, -plural]]
```

which means that `human` is true in the second model, `object` in the first and second and `plural` in the second.

It also means that the first three models are:

```
model([-human,+object,-plural])
model([+human,+object,+plural])
model([-human,-object,-plural])
```

This list can be very big. It is used by `encode_PropertyList/1` to encode properties as bit strings.

`encode_PropertyList/1` encodes the properties in `PropertyList` as bit strings as described by Mellish 88. The actual encoding is described in detail in Deliverable T1.6. Basically, if a literal is positive then it holds in the corresponding model and if it is negative then it does not hold in the corresponding model. 1's are used to represent the first case and 0's are used to represent the second. For example, if the list corresponding to human is `[-human, +human, -human]` then the corresponding bit string is `2'010`. Properties are stored in the database in the form `property_term(+Property,BitList)` and `property_term(-Property,NegBitList)` under the PImPLE key `&property_term`.

`encode_PropertyList/1` calls `encode_property_term(Property,PropertyList)` to encode a property `Property`. It in turn calls `bit_encode(PropertyList,BitList)` to encode the bit string for a property `Property` and `bit_neg_list(BitList,NegBitList)` to encode the negation of `Property`. These two predicates are defined in the file `bits.pl`. The method of encoding is described in more detail in the documentation for these two predicates. `encode_property_term` also calls `bit_zero_list(BitList)` to test whether the property is unsatisfiable (i.e., its bit string is zero). If it is, an error message is written.

The next section of `sort.pl` contains the predicates for compiling sort definitions. The predicates above determine the set of models implied by the axiom set and compile properties and negated properties but not the sort definitions themselves. We describe this here. The main predicate is `compile_sort(Sort,Axioms,NewAxioms)` which is referenced only by `compile_sorts/0` in `puser.pl`. We repeat the clause for `compile_sorts` here.

```
compile_sorts :-
  pimple_obj('&sort_definition',sort_definition(Sort,List)),
  compile_sort(Sort,List,Translation),
  svprops(Sort,Translation),
  fail.
compile_sorts.
```

`compile_sorts` compiles each sort definition. It nondeterministically chooses a sort definition from the database and then calls `compile_sort` to translate it into a bit string as described above. `Sort` is the sort name, `List` is a list of formulas or a simple formula

which comprises the sort definition and `Translation` is the translation of the definition into the set of basic connectives. `compile_sort` saves the compiled definition in the database itself. It then calls `svprops(Sort,Translation)` to save the translated sort definition. We describe `svprops` below (`svprops` is mnemonic for "save sort definition properties").

`compile_sort` first calls `translate_axioms0(Axioms,NewAxioms)` to translate the sort definition `Axioms` into the set of basic connectives. This predicate was described above and is also used by the axiom compilation predicates. The translated formula is `NewAxioms`. It then calls `bit_encode_axioms` with the translation `NewAxioms` to compile it into the bit string `Bits`. Finally, it calls `save_sort_bits` with the sort name `Sort` and the bit string `Bits` to save the compiled definition.

`bit_encode_axioms(A0,Bits)` takes a formula `A0` expressed with only the basic connectives and translates it into the bit string `Bits`. It does this in much the same way as `translate`. It rewrites the formula using `conj_from_list`, `implications_out`, `negations_in` and `cnf`. It then calls `expand_axiom_disjunction/2` with the rewritten formula to compile to the bit string `Bits`.

As described above, `expand_disjunction0` takes as input a formula and a partial model and produces as output a partial model containing all of the literals of the input model and possibly additional ones implied by one of the disjuncts of the disjunctive normal form of the formula. Therefore, if it is called with `[]` as the second argument, it will produce a partial model for one of the disjuncts of the DNF of the formula.

`expand_axiom_disjunction` works by producing a partial model for the input formula, calling `expand_partial_model` to "expand" the partial model so that properties which do not occur in any literal in the partial model have a variable in the "expanded" model where a corresponding literal would be and then finding the set of model numbers of models that the "expanded" model is consistent with by unifying the expanded model with each of the models. This set of model numbers `ModelSet` is then saved and the process is repeated for other disjuncts of the DNF. Finally, all of the model numbers are collected and sorted and `map_model_set/3` is then called with `ModelSet` and the number of models `NumberOfModels` to encode `ModelSet` as a bit string `Bits`. The main data structure is the clause `model_set_list(List)` which contains the set of model numbers found so far.

`expand_partial_model` and `expand_partial_model0/4` are tedious but require no comment.

`map_model_set/3` and `map_model_set0/8` map model number sets into bit strings. For each model number n in `ModelSet`, a `1` is put in position n in the bit string. For each model number not in `ModelSet`, a `0` is put in position n. Since machine words are not infinitely long, we have to split up the bit string into separate words. In PROLOG, this corresponds to creating a list of integers if the bit string is longer than can fit into a single machine word. For C-Prolog on a 32 bit machine, the restriction that we must use is 28 bits per integer since C-Prolog reserves 4 bits for tag information. The

number of bits per word should be parameterised to accomodate other PROLOGs and other machines but so far this has not been done in the code. It would be easy to do so however. `map_model_set0` takes care of splitting the bit string up appropriately and setting the 1 bits. It does this by bit rotating a 1 bit the appropriate number of places and then "logically or'ing" (the C-Prolog evaluable predicate \/) the rotated bit into the bit string constructed so far.

`save_sort_bits(Sort,Bits)` takes the sort name `Sort` and the bit string `Bits`, calls `bit_neg_list` to get the bit string for the negation of the sort definition and then stores the two compiled sort definitions under the key `&property_term` as terms of the form `property_term(+Sort,Bits)` and `property_term(-Sort,NegBits)` respectively.

The next section in `sort.pl` is the "Sort definition property section". This contains the definition of `svprops`. `svprops` is called with the sort name `Sort` and the translation of the formula into basic connectives `List`. It calls `svprops0` to collect all of the nondisjunctive parts of the translated sort definition into a list `PropertyList` which should then be a list of negated and unnegated properties and sort names. This is done so that the routine `u_sort_type` can omit mention of properties or sorts which are coextensive with some other sort and explicitly required in their definition. (Cf. discussion below.) Disjunctions are ignored because it would be too expensive to try to create a formula which describes the bit string precisely. Finally, `PropertyList` is saved under the key `&sortdefprops` in a term of the form `sortdefprops(Sort,PropertyList)`.

The next section of `sort.pl` contains the definition of `u_sort_type/2` and related predicates. It takes as input a bit string and returns as output the most specific list of negated and unnegated sort names which describes the bit string. This used to be called by `u_findunifs` in `upp.pl` as part of the UCG pretty printer for signs but this was very slow. Nonetheless, `u_sort_type` is very useful for debugging because it is basically the only way to make sense of the bit strings.

First, `u_sort_type` finds all of the negated or unnegated properties which subsume the bit string in question. It then calls `lowest_upper_bound_set/2` to try to reduce the set of property names by finding sorts which subsume the bit string and which imply properties in the set of properties which subsume the bit string. In other words, we try to replace properties with sort names which imply them. `lowest_upper_bound_set/2` calls `lowest_upper_bound_set/3` which calls `subsumes_no_member_of_set/2` which does the work of checking the set of properties against the set of sorts. This is where the `sortdefprops` terms come in. `subsumes_no_member_of_set/2` also guarantees that there isn't a more specific sort than the one found already in the set of sort names. The set of sort names is then put together by `u_sort_type1` and returned. I.e., the output is a list of sort names.

We'll now describe the predicates in `bits.pl`. It contains various utility predicates for creating and manipulating bit strings.

`bit_encode(List,BitEncodedList)` encodes the list of literals `List` as a bit string as described above. As with the code for sort compilation, it unfortunately assumes that

there are 28 bits available for bit encoding.

`bit_unify` unifies two bit lists. It does this by "logically and'ing" the two input lists pairwise and returning the output list. If all of the elements of the output list equal `0` then it fails. This means that the conjunction of the corresponding formulas is inconsistent.

`bit_subsume(X,Y)` succeeds if the bit list `X` subsumes the bit list `Y`. It does this by pairwise checking that the elements of `X` subsume corresponding elements of `Y`. `bit_s(B1,B2)` does this single element subsumption test. `B1` subsumes `B2` if `B2` equals `B1 /\ B2`.

`bit_identity` checks that two bit lists are identical.

`bit_neg_list` takes a bit list and creates the "logical complement" of it by taking the logical complement ("\") of each element of the input list.

`bit_zero_list` tests whether a bit list corresponds to `0` by testing that each element of the list is `0`.

`s_bintoatom` and `l_bintoatom` convert PROLOG integers to PROLOG-readable atoms representing the value of the integers as binary digits in binary reader format. `s_bintoatom` prints only as many digits as is necessary to represent the integer. I.e., it suppresses leading zeros. `l_bintoatom` does not suppress leading zeros. Both of these predicates are very useful for debugging the sort system. `s_bintoatom_list` and `l_bintoatom_list` convert lists of integers to "binary" atoms in "short" and "long" format respectively. `bintoatom_list` is equivalent to `l_bintoatom_list`.

```
?- s_bintoatom(15,X).
    X = '2'1111' ;
    no
?- l_bintoatom(15,X).
    X = '2'00000000000000000000000001111' ;
    no
?- s_bintoatom(32,X).
    X = '2'100000' ;
    no
?- l_bintoatom(32,X).
    X = '2'00000000000000000000000100000' ;
    no
?-
```

`type.pl` defines the predicates `u_ck_typ` and associated predicates, `u_mk_typ` and `translate_formula`.

The use of the sort system in PImPLE is to associate sorts with PROLOG terms. This association of sorts with terms is accomplished in an *association list (alist)* the elements

of which have the form `bits(Term,Bits)` where `Term` is the PROLOG term and `Bits` is the bit list. When two "sorted" terms are unified, their associated bit lists should be unified as well. This is usually done by appending two alists and then calling `u_ck_typ` to return a new alist.

`u_ck_typ` is a doubly recursive predicate which tries to unify the bit lists `Bits` of all terms `bits(Term,Bits)` which have the same `Term` argument. (PROLOG == is used to implement the identity test.) Bit lists are unified by calling `bit_unify` pairwise on corresponding elements of the bit lists being unified. The output alist that `u_ck_typ` returns contains only one `bits(Term,Bits)` for each PROLOG term `Term`. If the bit lists of two alist elements which have the same `Term` argument fail to unify then `u_ck_typ` fails. Thus, it implements unification of compiled sorts.

`u_mk_typ(Formula,Term,AListElement)` takes as input a PROLOG term `Formula` representing a sort formula, an arbitrary PROLOG term `Term`, compiles `Formula` into a bit list `Bits` and returns the alist element `AListElement` of the form `bits(Term,Bits)`. `u_ck_typ` calls `translate_formula` to do the compilation to bit list.

`translate_formula` takes a (list of) formulas and compiles it into a bit list. It calls `translate_axioms0` to translate the list of formulas into the basic set of connectives and then calls `bit_encode_axioms` to compile the translated formula into a bit list.

`unpack.pl` defines only the predicate `unpack/2`. It takes a bit list as input and returns a list of integers encoding model numbers for which a 1 bit was set in the bit list. This is very useful for decompiling bit lists in terms of model numbers for debugging the sort system.

There are also several utility predicates which are defined in `puser.pl` which are useful for debugging. They are discussed in Deliverable T1.6.

The definition of unification The most important definition in a unification-based system is obviously that of unification. In PImPLE, it is conceptually a triple,

```
unify(Term1, Term2, Result)
```

where each of the arguments is itself a triple, `Term:Constraints:Sorts`. Only for the first of these is PROLOG unification appropriate, as both `Constraints` and `Sorts` represent lists of restrictions on variables. To model unification in this context, we need routines that compare the restricted variables for identity, to check that the restrictions are consistent and to create a new representation of the restrictions corresponding to the conjunction of those on the input feature structures. So the resulting definition is

```
unify(Term:C1:S1, Term:C2:S2, Term:C:S) :-
        append(C1, C2, CUnchecked),
        check_constraints(CUnchecked, C),
        append(S1, S2, SUnchecked),
        u_ck_typ(SUnchecked, S).
```

In practice, this definition is not used, for reasons of efficiency. Instead, the call to `unify/3` is partially evaluated into the call in which it occurs.

`check_constraints/2` recurses through a list of pairs of variables and constraints. If a variable has become ground, its instantiation is check for consistency with the associated constraints. As we only allow disjunction and negation over atomic values, the tests we make are non-unifiability for negation, and membership of a list for disjunction. The current implementation is potentially incorrect in its treatment of such constraints, as it only verifies the satisfiability of a constraint when the variable to which the constraint is attached becomes ground. This means that a feature structure may be inconsistent, say by having two disjoint lists as possible disjunctive values, without causing a unification failure. This has not resulted in problems in practice.

The parser and generator The parser is that described in Deliverable T2.6. It operates in a strictly bottom-up fashion, computing all valid analyses of substrings. Intermediate results are stored in PROLOG's database, using as key the trailing vertex associated with a particular constituent. The routine that searches the database for appropriate edges may be compiled into the rules defined by the grammar writer. The basis of the generator supplied with the system is described in Deliverable T2.10 and in **2** below.

For reasons discussed in detail in Deliverable T2.6, we use a strictly bottom-up parser. The algorithm used by the parser has the following steps.

i Initially the value for currentvertex is 0.

ii Read the next word from the input buffer.

iii Find a lexical entry for that word, and its corresponding feature structure.

iv Add to the database the fact that the feature structure lies between vertices currentvertex and (currentvertex + 1).

v Choose a rule, and see if its right daughter unifies with the feature structure. If it does, work through the remaining daughters of the rule, checking that the required constituents can be found on the chart, returning the value of the left vertex of the left daughter. In the case of successful rule application and using the return value as the value of currentvertex and the feature structure of the mother of the rule, recurse to step iv.

vi Repeat step v until no rule applies. If there are other definitions for the current word, return to step iii.

vii Increment currentvertex. If there are still words in the input buffer, return to step ii, otherwise return the list of all feature structures lying between 0 and currentvertex.

Our implementation of this algorithm uses failure-driven repeats to enumerate possible lexical entries and to search exhaustively for possible rule applications. Note that this means that the parser can only handle grammars in which bottom-up rule application is guaranteed to terminate.

For reasons of efficiency, a further compilation of rules is performed. The description above assumes a matching function, as part of step iv, which compares elements in a rule with elements in the chart. This matching function can in fact be folded into the routine that accesses the chart. The routine which is responsible for this is called `sr_mk_rule/3`, taking as arguments the name of the rules, the mother feature structure and the list of daughters. The output of the compilation is a PROLOG clause under the head `&compiled_rule/8`. This procedure is called directly from the parser.

User interfaces PImPLE is started by invoking PROLOG with the saved state found in `${ACORD_HOME}/epi/Protolexicon/src/pl.ss`. The directory in which the system is started must contain, as a minimum, the start-up file, called `custom.q`. The format of this file is described below. After reading the start-up file, PImPLE will attempt to load the grammar files specified in the file.

Interaction via menus Interaction with the user is primarily menu-driven. At any stage, the following actions are valid.

`abort`	to abort current execution
`break`	to enter a prolog break
`halt`	to exit PIMPLE
`help, h or ?`	to give help information
`trace`	to start PROLOG tracing

The parser menu Once grammar files have been loaded, the parser menu is presented, indicated by the prompt `Sentence to parse>`. This menu has the options:

`redo`	to reparse previous sentence
`v`	to show variable bindings
`w Word`	to show feature structure of Word
`q`	to return to prolog
`quit`	to return to prolog
`String`	to parse String

where `String` is any string distinct from the previous options.

The analysis menu This menu has the prompt `examine results>` and the valid actions are:

`c`	to continue (return to parse level)
`d`	to show DAG structure of current edge(s)
`r`	to run back end procedure on output
`s`	to show statistics on current parse
`t`	to show parse tree(s) for the whole string
`t M N`	to show parse tree(s) for the string between M and N

The *start-up* file System parameters, such as the name of a grammar, the location of user-customization files and variables which control the behaviour of various routines, may be given in a start-up file. The name of the start-up file is `custom.q` and it must reside in the directory in which PImPLE is invoked. It consists of lines of the form:

```
set(Variable_Name, Variable_Value).
```

Variables The following variables and default values are used in the basic system:

Variable Name	*Value*	*Description*
pause	on	Wait for user input before continuing printing of output
symbols_are_categories	off	If symbolic names are used for elements of a grammar rule, insert that name as a value in the feature structure
path_for_category_symbol	cat	The feature under which the category symbol should be added
dummy_categories	[c,m,f,a,x]	Category symbols which may be ignored in rules
ask_for_replacement_words	off	If a word is unknown, ask the user for a replacement
include_lex_symbol	on	Add the string form of a lexical entry to the feature structure of a lexical entry
path_for_lex_symbol	phonology	The feature under which to insert the string form of a lexical entry
retain_file_information	on	Keep a record of the origin of objects
definition_file	.def	File extension for definition files
grammar_file	.gram	File extension for grammar rule file
lexicon_file	.lexical	File extension for lexical entries
sort_file	.sort	File extension for sort definitions
axiom_file	.axiom	File extension for axioms
top_level_goal_is_template	NULL	The template with which all legal covering edges should be compatible
last_sentence		The string form of the last sentence parsed
sort_edges_by_complexity	off	Compare all parses using a complexity metric
source_directory	$ACORD_EPI/Protolexicon/src/	Directory where source files are to be found
network_file	.net	File extension for the Protolexicon network
lexical_rules_file	.lr	File extension for lexical rules
morph_tables_file	.mt	File extension for morphological tables file
lexical_definitions	.lex	File extension for lexical entries
template_definition_file	.def	File extension for template definitions
unification	term	Type of unification used by the grammar
grammar_name	proto	The file name which, with suitable extensions, defines the grammar
grammar_directory		The directory in which grammar files are located
add_vertex_information	on	Add the vertex number to the feature structure of a lexical item
path_for_vertex_information	occnumber	The path under which to add vertex information
variable_aliases		Symbols to substitute for sort definitions in print routines
rules_are_compiled	yes	Are grammar rules to be interpreted or compiled
debugging_level	3	Amount of debugging information to be provided

Variables are implemented using database assertions of the form:

```
global_variable(Name, Mode, Restrictions, Value)
```

where Name is the variable name. Mode is one of system, user, once or fixed, indicating whether the variable is private to the system, user-definable, definable once only in a given session, or not redefinable. Restriction may be the word binary, any or some PROLOG predicate of arity one, such as integer/1 or atomic/1. binary restricts the possible values to on or off. The value any allows an arbitrary PROLOG term to be the value of a variable. The routine set/2 warns the user if a new variable is being defined or if a value is out of the range defined by the named predicate.

User-called routines The following routines may be called by the user to load or reload parts of the grammar.

`load_axioms`	
`load_sorts`	
`q_load_network`	
`q_load_templates`	
`q_load_lexical_rules`	
`q_load_tables`	Protolexicon morphological tables
`q_load_lexicon`	
`load_grammar`	
`q_load_and_close_lexicon(File)`	

With the exception of the last form, each predicate will load the file whose name is defined as the concatenation of the grammar name with the file extension appropriate for the file type. Each of them may also be called with a single file name argument, requesting the loading of that file. For discussion of the last predicate, see **6** below.

1.3 Relations to Other ACORD Components

The PImPLE system is unlike other ACORD components in that most of its functionalities are not used in the ACORD demonstrator. Instead the system is used to provide a "dumped" version of a particular grammar, which does not contain the information about the origin of objects necessary for the development version of the system. The procedure which produces a run-time version of the system is called dump/1 which takes a file name as its argument. It writes to that file PROLOG commands to add grammatical information to the database. This allows the file to be read in by means of PROLOG's consult/1.

The system used at ACORD run-time is a subset of that described above. It is defined by the files referenced in $ACORD_EPI/proto.pl, which also includes the repeated definition

of some predicates to avoid loading all definitions from a development system file, when only a small number of them are required.

Interfaces with other parts of the ACORD system In order to communicate with the other components, three predicates are defined. These are `nl_known_word/3`, `nl_parse_sentence/3` and `tg_new_generate_sentence/3`.

`nl_known_word(Language, List, Difference)` succeeds if, in the language `Language`, the list `List` begins with a sequence of tokens that represent a valid lexical item. `Difference` is the difference of `List` with respect to those tokens.

`nl_known_word(Sentence, Language, Result)` succeeds if `Sentence` is a grammatical sentence in `Language`, in which case `Result` is instantiated to the corresponding interface representation, including the sentence's translation into InL.

`tg_new_generate_sentence(Language, SynInL, Sentence)` succeeds if `SynInL` generates `Sentence`.

These predicates are defined in the file `$ACORD_EPI/sys_nl_eng`.

Mapping between PIMPLE and ACORD representations Given the different styles of representation used by the ACORD system as a whole and by the English and French grammar, in respect of sortal information in the first case and of semantic representation in both cases, we provide a mapping between the two representations. The mapping is relative simple, as there is an embedding of the ACORD sort system into the English sort system, and a simple isomorphism between the semantic representations. These are computed by the routines called `dm_translate/2` in the case of English and by `fdp_dm_translation/2` in the case of French.

The routine performs a analysis of the output of the parser, determining first of all the sentence type, as an assertion or a variety of questions. It then recursively translates from UCG structures into InL. The translation is primarily data-driven. Two points have to be handled with care. These are the construction of appropriate sort and gender information from the UCG sort system, a task performed by `dm_trans_index/6`, and the mapping between the unstructured representation of adjunct ambiguity of UCG and the structured form required in InL. The latter mapping is defined by the predicate `epi_dm_adjuncts_correspondences/2` and associated routines.

1.4 Potential for Development

Use in other situations As PIMPLE has been designed as both a development and run-time system, it is well suited for grammar development in other contexts, and has been so used for the development of alternative grammars of French, and also for

producing a grammar of Italian. The system produces, in conjunction with the routines described in the above paragraph, output suitable for use with the ACORD Resolver, and it has been extensively tested in this configuration. The environment that defines the necessary interface predicate is contained in `$ACORD_EPI/Dm_output/dm_resolve.pl`. The Resolver is invoked by using the `r` command at the Analysis Menu level. The variable `back_end` should be set to `dm_output` in this case.

Implementations in other languages As PImPLE is designed around the mechanism of term unification provided by PROLOG, it is likely that reimplementation in some other programming language would prove time-consuming.

Chapter 2

General Architecture of Generation

Dieter Kohl
IMS,
Agnès Plainfossé
LdM,
Mike Reape
ECCS,
Claire Gardent
CLF, ECCS

The general architecture of the generation process in ACORD and the modules shared by the three languages are described in this chapter. The language specific generators are described below; for English, see 3.2.2 ; for French, see 4.2.6 ; for German, see 5.1.5, 5.2.8, 5.2.9.

2.1 Scientific Background and Development

In text generation, the main emphasis has been on modelling world knowledge and knowledge about the user (e.g., his/her beliefs, goals). Recently however, there has been some interest in addressing the more *algorithmic* problem of generating a string from a semantic representation according to the syntactic, semantic and morphosyntactic constraints encoded in the grammar. Computational linguists have begun to investigate *abstract generation algorithms*, i.e., to develop generation algorithms for well-defined classes of grammars which can be shown to be *correct* and *complete* with respect to that

class of grammars. By *correct* we mean that a generator will not assign a string to a semantic representation which is not logically equivalent to a semantic representation assigned to the string by the grammar. By *complete* we mean that the generator will assign a string to every semantic representation which is logically equivalent to the semantic representation of some string in the language.

This recent interest was not present in the literature prior to the start of generation research in ACORD, which began in 1987. Only for LFG there was a paper on structure-driven generation from f-structures by Jürgen Wedekind (see Wedekind 86). This model provides the theoretical basis of the LFG generator used in the ACORD system.

However in 1988, three papers (Dymetmann and Isabelle 88, Shieber 88, and Wedekind 88) were published on the subject of generation from a semantic representation. Jürgen Wedekind's paper presents a structure-driven generator for LFG. This model extends the generator described in Wedekind 86 and is a seminal paper in the field. The two other papers are instances of the top-down and bottom-up approaches to generation respectively.

Until the beginning of 1989, the three language dependent generators were developed under the restrictions imposed by the first architecture. This lead to general algorithms and their implementation for generation in the LFG framework (see Momma and Dörre 87) and in the UCG framework (see Calder and al. 89). The language dependent generators which were developed within ACORD are described in more detail in Deliverable T2.10. The generators used in the final version of the ACORD system are described in more detail in the language specific parts of the technical documentation.

Despite the original contributions to the literature on the basic generation problem made by initial ACORD generation research, it was felt at the review in February 1989, that several improvements were necessary to make generation satisfactory within the ACORD system.

In the first versions of the generators InL is used as input. In the first architecture the language specific generators reside in a PROLOG process independent from the parsers and the DM. The InL provided by the DM to the generators was the resolved InL produced from the query, where the KB answer, after having been transformed into an InL format, is included.

While the UCG based parsers for English and French directly produce InL, the LFG German parser produces f-structures and needs a separate module to build an InL from an f-structure. For the generators this means there is an algorithm for UCG based grammars to generate from InLs, while for LFG there is a general algorithm only for generating from f-structures. So for LFG we also need a module which generates f-structures from an InL (see 5.2.8). Unfortunately, this module (called the f-structure generator) is not general since it depends too much on special meanings of partial InLs. Nevertheless, the development of the f-structure generator module has been very useful for experimenting the shortcomings of the architecture and for designing a new one.

The architecture used until 1989 has some principle restrictions:

- First, the generators did not have any access to information concerning the dialogue history. Since a resolved InL was the only information to generate from, the generators could only use some poor heuristics to introduce pronouns in the appropriate places or to select a definite article.

  The response of the system was in most cases a kind of repetition of the query transformed into an assertion and merged with the answer, in which it was likely that this repetition provided an inadequate pronominalization.
- Second, resolved InL as the only source for generation is simply underspecified with respect to how to say something.
- Further, noncanonicity of InL, and the encodings of some phenomena concerning plurals from the syntactic point of view, had been a problem for all three generators with respect to their robustness and efficiency.

  A formula of a semantic representation language is said to be canonical with respect to a particular grammar G if it can be derived by G. Although the first versions of the generators could be shown to be complete on canonical input, they could fail on noncanonical input.

  While the InL produced by any of the three parsers could be expected to be semi-canonical, an InL expression provided by the DM could not, since the DM did not have any information about canonicity.

  Semi-canonicity means that the structure of the InL mirrors something of the syntactic structure of the surface sentence, and that partial InLs are arranged in very restricted ways. Thus semi-canonicity could be only defined with syntactic terms.

We therefore decided to change the architecture with the goals to have a better quality response and more robust generators. To increase the robustness we either had to change our language dependent generators to treat noncanonical InLs, or to find a way to produce canonical semantic structures, other than InL. However, if a move was to be made towards a capability of generating arbitrary phrases defined by the grammar, then the noncanonicality limitations would have had to have been eliminated as it is totally unreasonable to expect that any application which produces input semantic representations to be generated, should have any knowledge whatsoever about which semantic representations (of all the wellformed ones) are canonical with respect to a particular grammar.

First, to increase the quality of the output the generators needed much more information in their input structure than provided by a resolved InL. While it was known that unresolved InL contains enough information to specify NP output, we still had to face the other InL deficiency mentioned above.

Second, reasonable NP planning had to be achieved, or at least a start made toward the development of an architecture that would support NP planning. Every language

provides several ways of expressing the same meaning. The possibility of introducing a referential expression either as a definite or indefinite description, proper name or pronoun whose interpretation depends on the local and global context results from linguistic abilities which allow a speaker to express himself/herself in the manner most appropriate to the context and interlocutor of the dialogue. One of the general strategies used to optimize communication is to enhance understanding by conciseness. In the context of natural language generation, the use of such strategies and linguistic knowledge is called *planning*. One very important type of planning is *NP planning*, that is, determining the form a particular NP is to be realized as. Successful NP planning depends on the dialogue history as well as the KB to find the most appropriate realization of NPs appearing in an answer.

NP planning plays a major role in generation. A generator without NP planning can lead to nonsensical dialogues such as:

- which driver goes to Paris?
- #he goes to Paris

or verbose answers such as:

- who goes to the city where truck1 delivers three computers ?
- #Mike goes to the city where truck1 delivers three computers.

Third, and most importantly from the point of view of the ACORD system, all of this had to be achieved in a language and grammar formalism independent way.

The solution developed in the second version of the ACORD generation component relies on two key ideas.

First, planning and generation are modular. That is, the generation and planning components are relatively independent and grammars are unaffected by any generation or planning requirements. This is in contrast to most of the planning-based generation represented in the literature. Indeed many of the proponents of such approaches would argue that this requirement is in fact impossible and that realistic planning requires integration of linguistic knowledge throughout a successful generation architecture. However as already mentioned, the ACORD generation components are required to satisfy some rather unusual constraints, the primary one being that any such planning-based architecture be able to treat English, French and German equally well. This fact alone dictates against an integration of linguistic knowledge in the planning component and suggests an architecture where all planning and phrase generation is completely modular. Therefore, our first concern was to develop a planning component that would be independent of any particular language, grammar or grammar formalism and that would make appropriate decisions on the linguistic realization of NPs. Some theoretical work on the problem of NP planning within ACORD we could use was described in Reape and Zeevat 88.

Second, the basis for phrase generation is an abstract representation in which both syntactic and semantic information can be encoded. This is essential in order to be able to convey to the phrase generators which bits of semantics correspond to which NPs, etc. Since InL did not seem to be a good choice for this task we defined a representation called SynInL which overcomes all the disadvantages of InL for planning and generation. SynInL is strongly inspired by $\overline{X}$ theory and the *D-structures* of Government and Binding Theory. As a result it abstracts away from the idiosyncracies of language particular syntactic structure and specifies a level of *abstract syntactic structure* that is *language independent.* Briefly, there are four types of SynInL formula corresponding to the four basic constituent types of $\overline{X}$ theory: *specifiers*, *heads*, *complements* and *modifiers.* SynInL is designed to allow the encoding of head-complement-adjunct dependencies and to indicate the decomposition of the input semantics to be generated in terms of these abstract constituency trees in a way that is language and formalism independent.

However, there is no indication of surface syntactic structure, word order or the relative scope of quantifiers arising in either complements or modifiers. The language generators are free to realize scope, surface syntactic structure and word order in any way which is consistent with the SynInL specification.

The reader might object to this elimination of scope distinctions. However, within the current ACORD architecture, any scope distinctions which are produced by the individual grammars or as a result of some semantics construction process are in fact artefactual. Furthermore, it might reasonably be argued that it should be possible to generate all possible scopes. This is typically done with *quantifier shifting* rules. Our solution is simply not to specify scope. This is another way in which we facilitate language independent generation.

The use of such an abstract representation contributes to satisfying the three requirements mentioned above as follows. First, most instances of noncanonicality are eliminated because subformulas are associated directly with syntactic constituents. Furthermore, the SynInL specification is itself a kind of normal form for semantic representations. Second, this intermediate representation mediates the linguistic decisions made by the PLANNER to the generators thus permitting effective NP planning, i.e., the SynInL encodes the plan. Third, since the level of representation is neutral with respect to languages, grammars and grammar formalisms, it allows for language and grammar formalism independent generation.

Since at this stage of the ACORD project it was not reasonable to change the output of the parsers from InL to SynInL, we had to think whether we could transform any InL produced by one of the three parsers to SynInL by a separate module. The f-structure generator already internally used a representation similar to SynInL to build f-structures. This internal structure could be build from any InL produced by the three parsers[1]. So it was clear that we could also transform semi-canonical InL to SynInL.

[1]The German generator was also used to answer queries given in French or English, for as long as there was no generator available for these languages.

It was then possible to set up the new architecture with SynInL as input to the language specific generators.

The new modules reside in the DM process, the only place where there is enough easily accessible information for planning.

The particular dialogue situation given in ACORD, together with knowledge sources primarily not designed for generation, means the query remains the main source to determine the structure of the answer.

Therefore, one input for the PLANNER had to be a SynInL which roughly reflects the syntax of the original query, and what had been understood by the system. This SynInL is determined by transforming the unresolved InL and the resolved one into SynInLs, which are matched against each other to get a SynInL without disjunctions.

The other input for the PLANNER is the KB answer and the language in use. The language is needed to generate appropriate gender information.

The PLANNER has to produce :

- the SynInL expression for the answer
- a list of possible objects to highlight on the screen

The output of the PLANNER is also used to update the dialogue history within the resolver to allow referring to objects mentioned in the answer using pronouns or definite descriptions.

The resulting system is a generation component which, as required, covers 3 languages, 2 grammar formalisms and 3 grammars while still allowing for NP planning. Note that the modularity of the global architecture means that the planning component could be extended to account for a more extensive set of linguistic phenomena, e.g., verbal ellipsis and anaphora.

As required by the comments of the reviewers at the end of 1988, the second version of the generation module constitutes an improvement on the first one in two basic ways. First, it is more robust in that the language dependent generators have to deal with canonical input only. Second, it produces better quality output by allowing for NP planning. A more detailed description of the evolution and functionality of the generation component within ACORD can be found in Deliverable T2.10.

2.2 Technical Description

2.2.1 Software and Hardware Requirements

The modules described in this section are all written in C-PROLOG. None of the modules are tested with other PROLOG dialects.

The code size is about 350 KB. All modules reside in the directory

$ACORD_STUT/planner

The code uses different prefixes.

inl_ for an InL checker

inl2syn_ for the InL→SynInL module

synunify_ for the matching of unresolved and resolved SynInLs

merge_ for the merger

plan_ for the PLANNER

Even in C-Prolog the common modules are fast. The only event which could make the PLANNER slow down a bit, is asking the KB for a definite description.

2.2.2 Common Modules

All modules are loaded into the DM process InL→SynInL.

This module consists of the files :

- basics.pl
- close.pl
- inlcheck.pl
- prime.pl
- rules.pl
- synunify.pl

It can be loaded alone by consulting

It is automatically loaded when loading the PLANNER.

Although **synunify.pl** is not needed for transforming InL to SynInL, it is part of the module, since it must be available to match unresolved InL and resolved InL.

The two main calls are :

inl_to_syninl(InL,SynInL)
synunify(SynInL_unresolved,SynInL_resolved,SynInL)

The transformation of InL into SynInL is based on the assumption that the InL provided is the output of one of the three parsers. This means that the InL is in a semi-canonical form i.e. the possible permutations within the InL are restricted. Making this assumption, the search space is fairly small, and the transformation is fast.

The basic strategy is to distinguish between *structural predicates* and *prime predicates* within an InL expression. Structural predicates are mainly **and**, **d**, **set**, and **imp**; prime predicates, all those which correspond to verbs, nouns and adjuncts on the surface level. The **occ** predicate, used only in unresolved InL, can be treated as a prime predicate as well.

The structural predicates have in common the fact that they have some InLs as arguments. Prime predicates always have a direct mapping to a partial SynInL. As already indicated in the list above of possible prime predicates, each prime predicate is classified to correspond to a basic syntactic category, where adjuncts do not distinguish between prepositions, adjectives and adverbs.

Using the distinction between structural and prime predicates, the transformation can be performed recursively.

1. If the partial InL is a variable or a dummy, the resulting SynInL is this InL.
2. If the partial InL is a structural predicate, get the transformation of its sub-InLs, and then apply a rule corresponding to the structural predicate on the SynInLs of their sub-InLs to get the resulting SynInL.
3. If the partial InL is a prime predicate, classify this predicate and create a SynInL for this prime predicate.

For the classification of the predicates an InL checker is used, which has the side effect of detecting serious incompatibilities between the parsers.

For the details of the rules see **rules.pl**. The principle strategy of the rules is to search for the correct relation between two SynInLs depending on the given structural predicate.

The possible relations are:

- both SynInLs can be unified
- one syninl is an argument of the other SynInL
- one syninl is an argument of a modifier of the other SynInL
- one syninl is a modifier of the other SynInL

By the assumption made above the module can expect that the information is given in a way that the head-argument-modifier structure is always well enough determined, to be able to decide where to put an argument or a modifier. In other words, the case where one SynInL is an argument of a modifier of the other SynInL, but this modifier is unknown at the time of the search, cannot occur.

The module cannot be used to produce InL from SynInL.

Unification of resolved and unresolved SynInLs While unresolved SynInL contains enough syntactic information to reconstruct to a high degree the original sentence, resolved InL does not contain any disjunctions. Since the resolver does not rearrange the structure of the InL, but only deletes or introduces some information, the resulting SynInLs are guaranteed to have basically the same structure. So the matching basically uses the following rules:

1. Two substructures can be unified with PROLOG unification.
2. The SynInL of the resolved InL is more specific than the one of the unresolved InL. Use this specific part and match the rest.
3. The SynInL of the resolved InL contains more information than the original SynInL. This information can be ignored, because it has only been introduced for technical purposes in a KB query.
4. The SynInL of the unresolved InL has a different specifier than the SynInL of the resolved InL. Since the specifier from the unresolved InL is the correct one, use it.

Merger and PLANNER During the development of the planning component, we first planned to have a merger module which calls an NP-planner. As it turned out, this approach was not appropriate, since the PLANNER has to be able, at each stage of the planning, to call itself recursively on sentence planning and adjunct planning as well. So all the merger has to do is to instantiate the planning process according to the type of question and the answer of the KB. In case the PLANNER has not been able to produce a SynInL, the merger also has to produce the appropriate scanned text indicators. Doing so, the merger never fails.

The PLANNER is loaded into PROLOG by consulting

planner_load

The whole planning process is started by:

merge_and_plan(SynInL_Orig,KB_Answer,Language,SynInL_Answer,Highlights).

Where

SynInL_Orig is the SynInL corresponding to the query

KB_Answer is the result from the KB

Language is the target language (usually the language of the query)

SynInL_Answer is the SynInL to give to the language specific generator

Highlights is a list of terms, which should be highlighted, if they refer to objects visible on the screen.

The merger treats three types of queries:

- yes/no questions
- wh-questions (e.g., *who, what, where,* etc.) and
- hm-questions (i.e., *how much* and *how many*).

For each type of query, there is a finite range of possible answers of the KB.

QUESTION TYPE	KB-ANSWER
yes/no question	`no(just,`*Number*`)`
yes/no question	`no`
yes/no question	`yes(even,`*Number*`)`
yes/no question	`yes(exactly,`*Number*`)`
yes/no question	`yes`
wh-questions	`no`
wh-questions	List of KB elements
hm-questions	`no`
hm-questions	*Number*
hm-questions	List of measure answers

A measure answer has the form:

measure_answer(*Number*,measure(_,*Unit*,*Number*))

Currently the PLANNER is only able to treat a list with one element.

The PLANNER has to be able to do two types of replacement, to give the correct answer:

- replacement of a numerical specifier by a new numerical value. This also includes the measure-case, where the numerical specifier consists of a unit and a number. This is needed for the answer of *hm*-questions and some answers of *yes/no*-questions.
- replacement of discourse referents as arguments of verbs and adjuncts. This is needed to answer *wh*-questions.

The list of KB elements always contains at least one KB representation of the discourse referents. For the PLANNER it is essential that this representation allows the reconstruction of both types of discourse referents used in InL: those which are used as arguments and indices of prime predicates other than `occ`, and those which are used as index of the `occ`-predicate and first index of the `set`-predicate.

The latter type is always needed for accessing global antecedents, while the first type is needed for the correct argument binding inside a SynInL.

A KB element can have one of the following forms:

obj(Id,[B1,B2,B3,B4],Class,Ref)	a KB object
duble_answer(Obj,Adjunct)	for answers to questions about prepositions
card(Obj,Number)	the object needs a number instead of a specifier
adj(Name,Obj_1,Obj_2)	the representation for adjuncts
fhg_is_equal(Obj,Obj)	behaves like Obj
rel(Name,Obj_0,Obj_1,Obj_2,Obj_3)	for relations. Instead of KB-Objects `0` could be used.
conjoin(<List of KB elements>)	conjunction of elements
disjoin(<List of KB elements>)	disjunction of elements

While `rel` is used only as an answer of an internal query about definite descriptions, the other forms are possible answers of a *wh*-question.

For each KB element, there exists at least one corresponding SynInL form. To avoid errors in the pronominalization process, the SynInLs for the KB elements are not planned until the first occurrence of the discourse referent in the focus of the question is found.

For more information about details on PLANNER see Deliverable T2.10. For an overview of the content of the single files for the PLANNER see the **Readme** file in $ACORD_STUT /planner.

2.3 Relations to Other ACORD Components

As already mentioned, the modules are loaded into the DM process. For this the file

sys_plan.cpr

is consulted. It differs from the loadfile mentioned before, in that it uses environment variables instead of absolute path names. See below in **2.4.3 Possible Combinations with other Components of the System.**

2.4 Potential of Developments

2.4.1 Development in the System

For adding new prime predicates, it is necessary to add these predicates to `inlcheck.pl`. In the case of new structural predicates or new special forms, it is also necessary to add more rules in `rules.pl` and `basics.pl` or `prime.pl`.

The PLANNER does not currently treat the copula in particular, and for more complex NP coordinations the result is far from optimal. Pronominalization of events or adjuncts is not possible as long as there is no representation in SynInL. The rules for pronominalization also need more refinement.

For the treatment of tense and other phenomena, the specification of SynInL must be extended.

The treatment of 'no' answers could be more sophisticated, to give cooperative answers. This needs more interaction between the KB and the PLANNER than now.

2.4.2 Development as an Independent Tool

The common modules for generation are obviously dependent on the specification and the expressive power of SynInL. Notice that the architecture for generation as it is used now together with the specification of SynInL, was defined and implemented within six months. This means neither SynInL nor the PLANNER are complete. The members of the generation task force will try to develop the given approach further within other projects including generation.

To make the PLANNER an independent tool, all dependencies of the particular representations used in ACORD must be evacuated from code. The SynInL has to become more powerful so as to carry all the information needed by language specific generators.

In the long run, the PLANNER should also be able to do sentence planning independently from the form of the query. This will be necessary for most types of cooperative answers.

The InL→SynInL module may be a useful basis for transforming a semantic representation into a structure which is closer to syntax.

2.4.3 Possible Translations into Other Languages

Except for the unification of SynInLs, none of the submodules are completely deterministic. There should be no great problems to use the code within other PROLOG dialects using the Edinburgh syntax. To rewrite the code for other languages similar to PROLOG should also be easy.

For algorithmic languages like C or PASCAL it will be easier to read Deliverable T2.10 and to implement the rules given there.

2.4.4 Possible Combinations with Other Components of the System

The InL→SynInL module which parses a sentence, tranforms the unresolved InL to SynInL and uses this as input to the generator, has also been used in local test environments for the three generators.

The PLANNER needs information from the resolver, the KB and the DM to produce its best results, but will also work without the databases of these components.

Using the PLANNER within another environment may require modifying the interface predicates given in `plan_inter.pl`. The predicates defined there are called to get

- global antecedents for pronominalization
- definiteness of objects
- definite descriptions
- the information whether an object is visible on the screen

The PLANNER is written primarily for the ACORD environment. Although we tried to write it in as modular a manner as possible in the given restricted time, it cannot be guaranteed that this particular version of the PLANNER will work in other environments without major changes in the code.

The PLANNER could easily be used for checking assertions, e.g.,

> User: the truck in Paris contains 10 pcs.
> System: Ok, truck1 contains 10 pcs.

Supposing *the truck in Paris* is resolved to *truck1*. The only thing that needs to be done is to add a rule for this case in the merger, which treats the 'ok' answer from the KB by running the PLANNER simply on the input SynInL, and returning an indicator for 'ok'[2]. In the 'no' case, the same answer strategy as for yes/no questions could be used.

[2]This indicator has to be added to the interface of the language specific generators as well.

Chapter 3

English Grammar ; Parser and Generator

Jonathan Calder,
Mike Reape,
Henk Zeevat
ECCS

This chapter describes the English Parser and generator, and their underlying UCG grammar. See above in 1 the grammar environment.

3.1 Scientific Background and Development

Our original intention for ACORD was to develop a GPSG fragment for English and combine that with a version of the Earley parser. By the start up time however, it had become apparent that this was not an optimal choice. Experience with GPSG had made it clear that a full implementation forces a departure from standard GPSG. An interesting development at the time was Pollard's first development of HPSG, where the massive set of rules in (fully expanded) GPSG was reduced to only 6 schematic ones (see Pollard and Sag 87 for later developments). Accordingly we tried to extend these ideas to incorporate a larger fragment and to include semantics. In the latter enterprise we found that we could achieve more simplicity by incorporating schematic type raising and categorial polymorphism. Only later we discovered that we were dealing with a polymorphic version of categorial grammar.

The resulting model UCG has been extensively documented in Zeevat et al. 87, Calder et al. 88, Moens et al. 89 and Deliverables T2.1 and T1.6. This last deliverable is an

almost complete documentation for the ACORD version of the fragment.

The grammar fragment was first developed on paper as a side product of generating the Deliverable T2.1. This made the first implementation on one of the earlier versions of the PIMPLE system (see 1) straightforward. Unfortunately we found empirically that the full Earley parser does not work for the class of grammars that contains UCG: polymorphism and the Earley type prediction are uneasy bedfellows. The replacement of the Earley parser by the current robust chart based shift reduce parser still seems the best solution.

Major later developments have been the incorporation of pronoun resolution, development and interfacing to a preposition disambiguator, the incorporation of a resolver interface, adopting a new sort management system, the development of a generator and finally adapting the grammar to the new treatment of plurals adopted by the ACORD project. The connection to the protolexicon has led to a somewhat different file organisation than the original one.

3.1.1 The Parser

The parser is the standard PIMPLE described above in 1.

3.1.2 The English Grammar

The English grammar is a Unification Categorial Grammar. It combines the syntactic insights of Categorial Grammar with the semantic insights of Discourse Representation Theory. The addition of unification to these two frameworks allows for a simple account of the interaction between different linguistic levels.

A concise introduction to UCG can be found in Calder et al. 88. The most complete documentation of it is given in Deliverable T1.6.

The only undocumented part of the current grammar is the change of the semantics of plurals to meet the ACORD specification. This change involves making a fundamental distinction between the semantics of singular and plural NPs, and forces a reduplication of a large part of the NP treatment. An example is the distinction between the two versions of the English determiner *the* whose definitions come out as:

```
the: [@'Determiner', @'Definite', :cat2:cat1:category:feature:number = sg].
the: [@'PluralDef'].
```

A new feature with possible values `sg` and `pl` has accordingly been introduced to make this distinction on nouns and NPs and a number of changes have been made to acco-

modate the syntax of the five place set predicate. An example is the following analysis of *three boys run.*

```
set(X,x,boy(x),run(e,x),3)
```

The appearance of the five-place predicate is typically dependent on the marking of the noun phrase in question as `pl`. The PIMPLE representation of the semantics of a plural NP is then:

```
...
( XX::['Plural']:
  X::['Singular]:
  1#( X:_:_):
  #2:
  Conjoin):
...
```

where `Plural` is the element of the sort system for plural objects, `Singular` likewise for singular, `#1` is the translation of the nominal in question, `#2` is the translation of the element with which the nominal combines and `Conjoin` is either a structure representing the semantics of coordinated expressions or an indication of the cardinality of the set. This form of semantics has to be associated with every grammatical object which gives rise to a plural noun phrase. These objects are templates for determiners, personal, possessive and anaphoric pronouns, cardinals, conjunctions, WH-elements and the plural bare noun rule.

The adoption of this representation of sets and plurals means that some analyses available to us earlier are no longer possible. A notable case is the adjunction of a modifier to a verbal projection that contains a plural.

3.1.3 The Generator

The English generator supplied with the system is more or less the same as the French and the German generator. The software component is documented in detail below (see **3.2.2**).

3.2 Technical Description

3.2.1 File Description

The files comprising the grammar are those defined by the PIMPLE and Protolexicon systems, with the variable `grammar_name` set to `proto`, and the variable `grammar_directory` set to `$ACORD_EPI/Lexicon`. These files reference a group of other files whose individual contents are described in the file `MANIFEST` in that directory.

3.2.2 English Generation

The files containing the code for the English generator are contained in the files in the directory `\$ACORD_EPI/Generation`. There are ten such files. They are described in more detail in the file `MANIFEST` in that directory.

```
README
compile.pl
dm_inv_prep.pl
genutilities.pl
interface.pl
lexical.pl
load.pl
match.pl
u_gen.pl
u_reduce.pl
```

We will describe the major functionality of each file here.

`README` gives a short description of each of the files in the directory.

`load.pl` loads the English generation source files, defines `u_show` and `u_produce` which strip the names of the rules from the phonology of the sign generated and returns the list of words and sets the `global_variables identity` to `on` and `u_show` to `off`. `identity` is a switch to allow processing of lexical entries with identity semantics (as described in Deliverable T2.10) and `u_show` is a switch which disables certain debugging information.

`genutilities.pl` contains various utility predicates either used by the generator or useful for debugging.

`u_listdb` writes a listing of the "compiled semantics" to a file. (The compiled semantics is described below.)

`u_writedb` writes a listing of the compiled semantics to the terminal.

u_erasedb erases the compiled semantics from the database.

u_verify is the standard verify predicate defined in terms of double PROLOG negation.

u_variants is the standard definition for testing alphabet variance in terms of the predicate **numbervars**.

dm_inv_prep.pl contains a set of unit clauses of the form **dm_inv_prep(Adjunct,UCG)** where **Adjunct** is one of the standard ACORD InL adjunct terms and **UCG** is one of **Adjuncts** possible translations into English prepositions. Unfortunately, this file has to be produced manually to prevent lots of infelicitous generations involving poor preposition choices.

interface.pl contains all of the interface predicates necessary to interface the English generator to the DM.

tg_new_generate_sentence(english,text(Marker),Sentence) is the top level predicate which is called by the DM. Its first argument is the name of the language (**english** in our case). Its second argument is the input to the generator. It is either a SynInL expression, a term of the form **text(Marker)** where **Marker** is a tag indicating that only canned text should be printed or a term of the form **affix(Marker,SynInL)** where **Marker** is a "prefix" to the SynInL formula to be generated and **SynInL** is a SynInL expression produced by the planner.

u_eng_local_translate_text_marker/2 translates a text marker into a list of atoms ("canned text") to be printed as the answer.

u_eng_local_select_prefix/4 deals with the **affix** and SynInL cases. If there is an affix, it translates it and returns it as the head of the list **Sentence**. It also returns the tail of **Sentence** as **RestSentence**. This is then passed to the generator predicate. It also returns **PureSynInL** which is the input SynInL to the generator.

u_generate/2 takes a SynInL formula as input and returns a list containing the sentence generated. It first translates the input SynInL formula into a form which is more convenient for the English generator by calling **dm_inv_syninl**. Mostly, this involves mapping from the InL representation for sorts and terms into the English UCG representation for sorts and terms. UCG sorts are of course represented with bit lists. A goal sign is then created for the generator with syntactic category **sent^fin** (i.e., a finite clause) and empty gap lists. The predicate **u_gen** is then called with the "inverted" SynInL formula, the goal sign and the input constraint and bit lists. The predicate **u_show** is then called with the goal sign (which has been instantiated by the generator) to extract the sentence generated from the **phonology** attribute.

If **tg_new_generate_sentence** fails to generate for some reason, it returns an apology to the user.

Most of the code in this file is based on similar code written by Dieter Kohl of IMS.

dm_inverse.pl defines **dm_inv_syninl/2** and related predicates. Mostly **dm_inv_syninl** maps the InL representation of sorts, terms and occlists into the corresponding UCG representation of these three types of subformula. The code is very implementation and grammar specific and is of no theoretical interest so we will not describe it further here.

We'll now describe the five main generator code files. **u_gen.pl** and **u_reduce.pl** define the major predicates **u_gen** and **u_reduce** respectively and related predicates. **match.pl** defines **u_get_sub_syninl** and related predicates. It identifies the complement or modifier SynInL subformula to be generated next on the basis of the subcategorisation requirements of the sign about to be generated. **compile.pl** defines **u_compile_semantics** and associated predicates which "compile" the semantics of lexical entries for efficient lexical access. **lexical.pl** defines the two major lexical access predicates, **u_lexical** and **u_identity**.

We'll now describe each of these files briefly in turn. The general approach of the code is based on the algorithm for UCG generation from SynInL formulas described in Deliverable T2.10. However, for efficiency some metaknowledge about the coverage and structure of the grammar has been folded into that algorithm essentially specialising some of the predicates. The major effect has been a great increase in the number of predicates defining **u_gen**, a specialisation of some of the clauses of **u_reduce** and a simplification of the definition of **u_get_sub_syninl**. The clauses in **compile.pl** and **lexical.pl** are fairly independent of the details of the algorithm and have not been affected by this programme transformation very much.

u_gen.pl defines **u_gen/6** and related predicates. There are many specialised clauses for **u_gen** but the final "elsewhere" clause best illustrates the basic idea behind the generation strategy. Roughly, **u_gen** implements the *top-down, predictive* half of the generation strategy. It is listed here.

```
u_gen(SynInL, GoalSign, C0, B0, C, B) :-
  u_syninl_ucg(SynInL,Sign,[Args,Adjs]),
  u_lexical(Sign, C0, B0, C1, B1),
  u_reduce_args(Args, Sign:C1:B1, [], Sign1:C2:B2, [], F1),
  u_reduce_adjs(Adjs, Sign1:C2:B2, GoalSign:C3:B3, F1, F),
  u_unfreeze(F,[]),
  check_constraints(C3, C),
  u_ck_typ(B3, B).
```

SynInL is the input SynInL formula to be generated. **GoalSign** is a partially instantiated sign constraining the way that **SynInL** is realised. **C0** and **B0** are the input constraint and bit lists. **C** and **B** are the output constraint and bit lists. This pattern is repeated over and over in all of the predicates so we shall not discuss or mention constraint or bit list variables any further.

First, the input **SynInL** is passed to **u_syninl_ucg** which maps **SynInL** into a partial English UCG sign **Sign** and a list **Args** containing the SynInL formulas of any

complements and a list **Adjs** containing the SynInL formulas of any modifiers or adjuncts. **u_lexical** is then called with **Sign** to unify **Sign** with a lexical entry. Then **u_reduce_args** is called with **Args** to generate the complements of the head **Sign** and then finally **u_reduce_adjs** is called with **Adjs** to generate the adjuncts of the head. (Of course, there is no guarantee that one can generate all the complements first and then the adjuncts. However, this is safe for the English grammar since all cases where this is not true are taken care of by specialised clauses of **u_gen**.) The process of generating complements and adjuncts typically "uses up" subcategorisations or otherwise modifies the category of the sign being worked on. In the end, after all complements and adjuncts have been generated the result sign must unify with the goal sign **GoalSign**.

One of the major problems with this style of generation is to map between the SynInL representation and the semantics of UCG signs. Both the SynInL representation and the UCG grammars are rather idiosyncratic and contain arbitrary deviations from the idealised presentation of Deliverable T2.10. Both **ugen** and **u_syninl_ucg** therefore contain a large number of very specific clauses which map the SynInL formulas into a form more convenient for lexical access. In fact, the strategy that we have adopted was to do anything necessary in these two predicates to allow lexical access to unify partial signs directly with lexical entries rather than being unified via a unification predicate defined within the generation code. In those cases where there is something very specific being done, there will be a clause of **u_gen** which usually does not reference **u_syninl_ucg**. In most other cases, there will be no corresponding clause of **u_gen** but there will be one for **u_syninl_ucg** which is used by the last "elsewhere" clause of **u_gen**. We'll only describe here those clauses which actually implement a different control strategy than that described in Deliverable T2.10. We will not describe those clauses which appear just to accomodate differences between the SynInL specification and the English grammar.

Perhaps the most significant case where there are specialised clauses of **u_gen** is to deal with verbs. The basic idea here is that we really only generate finite clauses and that we only generate VPs for auxiliaries and negation. We also get a specification in the SynInL of whether we should generate an active or passive clause. We treat the two cases separately since they are slightly different.

For the active case, we first map the SynInL **Sem0** into **Sem** with the predicate **u_inl_ucg**. **Sem** is compatible with UCG InL . We then create a partial sign **Sign** with semantics **Sem**. Next, we split the list of adjuncts **Adjs** into a list of negations **Negs**, a list of PPs **PPs** and a list of subordinate clauses **Conds** with the predicate **u_verb_adjuncts**. Next, we can decide whether we need to generate a finite verb or a base verb first on the basis of the presence or absence of a negation modifier. If there is no negation, then we generate a finite VP. If there is a negation, then we generate a base VP as that is what the negation 'not' subcategorises for. We then determine what the category of the verb will be by examining the number of arguments it has and by plugging in the verb feature **fin** or **bse** as just discussed. The next step is to call **u_lexical** to unify the partial verb sign with the lexical entry for some verb. We then create a new category **VPNoAdjsCat** and a partial sign with that category **VPNoAdjs** which subcategorises for a single complement (the subject) and has the same syntactic feature as discussed. This is used as the goal

sign in a call to `u_reduce_args`. `u_reduce_args` is given the argument list `Args` to generate and generates them all except the subject semantics `Subj`. It generates all the arguments except the subject thus creating a VP of the right syntactic type. We next create another VP goal sign of the same type `VPwithPPs`. This will be the goal sign in a call to `u_reduce_adjs` which generates all the PP modifiers `PPs`. We repeat the same process with goal sign `VPwithConds` and input `Conds`. Next, we must take care of the negation (if there is one). If there is a negation we call `u_reduce_adj` with the SynInL `Neg` to get back yet another VP `VPNeg`. We now need to generate a finite form of the verb *do*. We do this by creating an active sign `DoVP` that subcategorises for the negated VP, instantiating it to subcategorise for two complements and requiring that it be `fin`. We also apply this sign to `VPNeg` to create a mother sign `VP`. Since all of the forms of *do* have identity semantics, we then call `u_identity` to unify it with one of the lexical entries corresponding to the forms of *do*. If there was no negation then we were left with a finite clause anyway after generating all the adjuncts so in either event we now have a finite VP `VP` subcategorising for the subject `Subj`. Finally, we call `u_reduce` with `Subj` and `VP` and goal sign `GoalSign` which should succeed assuming that agreement, etc. is correct.

By taking over the control strategy in this clause we are able to severely reduce the search space involved in generating clauses and VPs. The `u_gen` clause for the passive is similar except that in the passive one always has to generate the copula and it has to subcategorise for a `bse` VP. We won't discuss the details here as they are very similar to the active case.

The case of nouns is similar. Since nouns also have adjuncts we also take over the control strategy here via a specialised clause of `u_gen`. Again, a partial sign `Sign` is created containing the appropriate semantics. Then `u_lexical` is called to unify one of the lexical entries with `Sign`. Next, `u_noun_adjuncts` is called to separate the list of adjuncts `Adjs` into a possible possessive "adjunct" `Possessive`, a list of PPs `PPs` and a list of relative clauses `RelClauses`. A noun goal sign `Sign1` is then created and `u_reduce_adjs` is called to generate `PPs` as `Sign1`. `u_reduce_relative_clauses` is then called with `RelClauses` to generate the relative clauses. A different predicate is used here since relative clauses are also handled separately to reduce the search space. Next, `u_reduce_adjs` is called with `Possessive` and the goal sign `GoalSign` to generate the possessive if there is one. It will always be the case that the possessive should be generated last and this clause of `u_gen` guarantees that that happens thus improving speed considerably.

`u_reduce_relative_clauses` generates a list of relative clause SynInL formulas one at a time by calling `u_reduce_relative_clause`. It first creates a goal sign `NMod`. It then creates a relative pronoun sign `RelPro` with phonology *that*. (We don't bother with *who*, *whom* or *which* since *that* is case, number and animacy insensitive.) We then apply the relative pronoun sign to a new sign `VP` to get the mother sign `NMod`. The result is that the relative pronoun gets further instantiated and the `VP` sign gets partially instantiated. In UCG of course, relative pronouns subcategorise first for a sentence to their right looking for an NP to its left (i.e., a VP) and then for a noun to their left. `RelPro` is now sufficiently instantiated that we can profitably do lexical access with it

with `u_lexical`. We next apply the modified sign `NMod` to the input noun sign `Sign` to get the result sign `NewSign`. It is then used as the noun sign for reducing the next relative clause. When all the relative clauses have been generated the noun sign must unify with the goal sign `GoalSign`.

This takes care of efficiently instantiating signs in such a way to reduce the search space of creating a noun modified by a relative clause but doesn't deal with generating relative clauses themselves. This task is split between the predicates `u_reduce_relative_clause` and `u_reduce`. The problem for relative clauses is that the `S/NP` to be generated may involve an extracted embedded subject or object. This can give rise to a large amount of search. What we try to do is minimise the search by determining whether there has been an extraction or not. If there has then we force an NP subcategorisation onto the gap lists to limit the search space. We will describe this in detail here.

`u_reduce_relative_clause` has two clauses. The first deals with the nonextraction case (i.e., where the NP subcategorised for is the subject of the relative clause) and the second deals with the extraction case. The first clause checks to see whether any of the complements to be generated is a relative pronoun by seeing whether a `PRO(rel(_),_,_,_)` specification is an element of the argument list `Args`. If so, then it simply calls `u_gen` with the clause SynInL and the input sign. It may guess wrong by finding an object that is actually the relative but this is unavoidable. There is no way to tell which is which. However, this isn't so bad as it eliminates all those cases where the extracted NP isn't the complement of the verb of the relative clause.

The second clause deals with the extraction case. First, it takes the input sign (i.e., the VP sign) and uses it as the mother in the `gapelim1` rule which eliminates a gapped NP on a finite clause. I.e., this rule is precisely the one used in relative clauses to take the extracted NP off the gap list and reintroduce it as a subcategorised NP. This produces a clause `Sent` which is then passed to `u_gen` along with the relative clause semantics `Clause`. In other words, we have predicted top-down that a gap introduction will be required of a relative pronoun somewhere along the line. `u_reduce` takes care of the gap introduction step. We will describe this in the section on `u_reduce`.

The `u_syninl_ucg` clause for prepositions is not exceptional except in that it references `dm_inv_prep` to translate the SynInL adjunct term into the `phonology` of an English preposition. The partial sign to be used for lexical access will have its `phonology` instantiated and the `predicate` of its `semantics` uninstantiated. This is about all that can be done since one cannot predict in advance what the form of the predicate will look like. Since we really only need to guarantee the correct string form, this solution seems adequate although very ad hoc.

A comment that should be made about many of the clauses of `u_syninl_ucg` is that they instantiate the phonology of the partial sign they create for lexical access. Due to an optimisation in lexical access, this guarantees very fast lexical access. Since the SynInL is very specific in some cases we are able to eliminate a lot of search in this way.

One place where we can eliminate a lot of nondeterminism is in the generation of phrases like *3 litres of wine*. These so-called "measure" words are of the schematic form `(C/C/np)/((X/X/np)/noun)/pp^of`. That is, the noun *litres* subcategorises for a case-marked PP with PFORM `of` and the cardinal itself. This is due to the peculiar InL semantics for these constructions. By specialising a clause of `u_gen` for this construction we can eliminate a lot of blind search.

First, a sign `Sign` for the head is created with the semantics of the "measure" noun. Then a SynInL formula `Cardinal` is created for the cardinal of the appropriate type. Its semantics involves a `set` predicate which actually specifies the actual `Number` of the cardinal. This is instantiated in `Cardinal` to aid lexical access. Next, `Sign` is unified with a lexical entry. We are now ready to consider the subcategorised `pp^of`. Although you can have expressions like *three litres of the finest red wine from Florence* the SynInL specification only allows the semantics of lexical nouns corresponding to the `pp^of`. Therefore, `u_gen` has to go through the work of type-raising the noun into an NP, combining it with the case-marking preposition *of* and then finally combining this with the measure noun. This is what we'll describe now.

A noun goal sign `NounSign` is created and then the semantics of the object of the PP is passed along with `NounSign` to `u_gen` to be generated. `NounSign` is then type-raised by one of the "barenoun" rules `barenoun_mass` or `barenoun_plural` into an NP `NP`. A preposition sign `P` is then created with `phonology of`. This is then used for lexical access by calling `u_identity`. This returns the case-marking version of the preposition *of*. We then further instantiate `NP` so that it is a sufficiently instantiated type-raised NP. We then apply `NP` to `P` to get a PP `PP`. We then apply `PP` to the measure noun sign `Sign` to get a sign `Sign1` of the schematic form `(C/C/np)/((X/X/np)/noun)`. We then apply `Sign1` to a new cardinal sign `CardSign` to get the result sign `GoalSign`. `GoalSign` is of course the noun sign. This application results in everything being coinstantiated properly. Finally, we call `u_reduce` with semantics `Cardinal`, sign `Sign1` and goal sign `GoalSign` to generate the cardinal.

The point to note is that since the order of generations is tightly controlled, the search strategy cannot go astray. Indeed, there is almost no search in this clause.

Finally, a few words are in order about coordination. Coordination does not use specialised clauses of `u_gen`. However, there is a certain amount of subtlely in the clauses of `u_syninl_ucg` for the conjunctions **and** and **or** and the comma. Basically, the idea is that if you have the conjoined NP *Jo, Henk and Klaus* then you get a right-associative structure where *Henk* and *Jo* combine via the *and* and then *Jo* and the coordination combine via the comma. If you add more commas then the structure repeats recursively. So the generation strategy is as follows. If the number of conjuncts to be generated is greater than two then we use the clause for the comma. It takes the first element `Arg1` off the list of arguments and creates another `G_JOIN` structure `Join`. It then returns a sign for , with phonology instantiated with `[Join,Arg1]` as the argument list. If there are more than two elements in `Join` then eventually the same process will be repeated. Eventually, `Join` will only have two elements and then the clause of `u_syninl_ucg` for **and** and **or** will be referenced. This just creates a sign for the conjunction and passes

the two argument conjuncts on. In general, `u_get_sub_syninl` cannot determine how to decide when to choose type-raised NPs to generate next. So, it just chooses *deterministically*. In general, this is not safe but it is okay for the current English grammar. However, since the sign for comma just happens to look to its right first for a coordination of NPs, if we put `Join` as the first argument in the argument list everything works out. In fact, the conjuncts get generated in the order that they appear in the argument list. This is rather ad hoc but adequate to the specification it deals with.

The rest of the clauses in `u_gen.pl` follow the general pattern, i.e., there are specific clauses for `u_syninl_ucg` and the last "elsewhere" clause of `u_gen` is used.

There are three major predicates defined in `u_reduce.pl`. They are `u_reduce`, `u_reduce_args` and `u_reduce_adjs`.

`u_reduce` takes three arguments. The first is the SynInL to be generated. The second is the sign to be reduced, i.e., the one doing the subcategorisation usually. The third is the result sign or the sign to be reduced to.

`u_reduce` has six clauses. The first clause takes care of the case where a relative pronoun is the subject of its relative clause. In this case, we just want to ignore the SynInL for the relative clause.

The second clause is the second half of the nonsubject relative pronoun case. If the SynInL formula is a relative pronoun specification then we try to apply one of the "gap introduction" rules `npgap`, `nppgap` or `ppgap` so that the input sign `Sign` contains the subcategorisation on its gap list and its daughter is the goal sign `GapSign` which introduces the subcategorisation for the relative pronoun. In this way, when we find the relative pronoun specification we know exactly what to do (since we have a constituent on the gap list) and we have a relative pronoun specification.

The third clause takes care of deleted complements. If the SynInL is the atom `unknown` then we know that the "delete optional complement" rule `optionalpp` has to be applied to eliminate a subcategorisation and instantiate its index to `unknown`. Therefore we just use this rule to reduce the goal sign `NewSign` to the input sign `Sign`.

The fourth clause is based on the original UCG generator. It takes care of the case where there is an argument subcategorised which is not `np^F` or `pp^F`. In this case, we apply the input sign `Sign` to an argument sign `Arg` to get the result goal sign `Result`. Then we generate the argument sign `Arg` with the SynInL `SynInL`.

The fifth clause is also based on the original UCG generator and takes care of subcategorised nps and pps. The np or pp is type-raised to a schematic sign `RaisedSign` of the form `C/C/np` or `C/C/np`. The type-raised sign is then generated by `u_gen` with semantics `SynInL`. Finally, `RaisedSign` is applied to the input sign `Sign` to yield the new sign `NewSign`.

The sixth clause implements an efficient, top-down approach to case-marking prepositions. Rather than hypothesize case-marking PPs bottom-up for every NP we just examine the current subcategorisation to check for case-marking PPs. The first thing is to see whether the subcategorisation is for a `pp^Pform` where `Pform` indicates a case-marking preposition. If so, we create a partial preposition sign `P` with phonology `Pform`. We then do lexical access (with `u_identity`) to get the lexical entry for the preposition. We then create a type-raised NP `NP` and apply it to the preposition to get the PP `PP`. We then apply the PP `PP` to the input sign `Sign0` to get the result sign `Sign`. Then the only thing left to do is generate the NP `NP` with semantics `SynInL`. This is very efficient as it is totally predictive.

`u_reduce_args/6` takes as input a list of SynInL expressions `SynInL`, a sign to be reduced `Sign` and a goal sign `GoalSign`. The base clause succeeds if the input sign unifies with the goal sign. If so, it returns the list of arguments `Args`. This is crucially used by the clauses of `u_gen` which deal with verbs. In other words, unification of the input sign and the goal determines success, not the exhaustion of the argument list.

The second clause calls `u_get_sub_syninl` to choose an element `SubSynInL` of `SynInL` according to the type of `Sign` returning the rest of the list `NewSynInL`. It then calls `u_reduce` to reduce `Sign` to `NewSign` with semantics `SubSynInL` and then recursively calls `u_reduce_args` with `NewSynInL`, `NewSign` and the goal sign `GoalSign` returning the argument list `NewArgs`. In other words, `u_reduce` doesn't have to decide what an argument's semantics is. This is done before it gets called. This allows a general treatment of those cases where it is clear what semantics goes with which syntactic complement.

`u_reduce_adjs/5` is similar except that it has no reason to return a list of unreduced adjuncts and calls `u_reduce_adj` instead of `u_reduce`. Also, `u_get_sub_syninl` is not used to decide which adjunct to reduce. Instead, the head of the list is always chosen. It calls `u_reduce_adj` instead of `u_reduce` for two reasons. First, `u_reduce` is unnecessarily general for adjuncts. Second, there are only two cases to consider for adjuncts and they are both special to adjuncts.

Therefore, `u_reduce_adj` has two clauses. The first takes care of PPs which are always introduced by the "optional PP introduction" rule. First, a sign `Sign` is created which subcategorises for a `pp`. `Sign` is then used as the mother in a unary rule application where `Sign0`, the input sign, is the daughter. In other words, `Sign0` gets a PP subcategorisation added to it. The subcategorisation for the `pp` is then type-raised to the form `C/C/pp` and applied to `Sign` to get the new result sign `NewSign`. Finally, the PP is generated with semantics `SynInL` as the type-raised PP `RaisedSign`.

The second clause is very simple and treats "lexical adjuncts", i.e., those which are functors over their heads. In this case, a sign `AdjSign` is created by applying it to the input head sign `Sign` to get the result sign `NewSign`. `AdjSign` is then generated with the input semantics `AdjSynInL`.

It should be noted that both `u_reduce_args` and `u_reduce_adjs` are deterministic. This was not possible with previous versions of the generator. This has only been made possible by making some of the control explicit in `u_gen`.

`match.pl` contains the definitions of `u_get_sub_syninl` and related predicates. As we have seen, `u_get_sub_syninl` chooses which argument corresponds to the subcategorisation about to be reduced. It begins by trying to find the *semantic index* of the subcategorised constituent. It then calls `u_match` with the syntactic category of the argument and the index.

`u_match` has four clauses. It takes the syntactic category of the argument `Cat`, its semantic index `Idx`, an input argument list `Args` and returns the semantics to be generated `Arg` and the rest of the argument list `NewArgs`.

The first clause treats noun arguments. In this case, there should only be one argument semantics, that of the noun.

The second clause treats deleted complements. In this case, the term `_\unknown` is simply deleted from the list of argument semantics.

The third clause deals with subcategorisations for type-raised NPs. (These are introduced by both coordination and "measure" nouns.) Here it just nondeterministically removes one of the argument semantics since it has no basis on which to choose between them. This could stand a lot of improvement.

The fourth clause attempts to find a SynInL expression which has the same index as the argument. It does this by calling `u_rm`. `u_rm` goes through the list trying each one out by calling `u_get_arg_syninl`. This predicate is needed to pinpoint the index in the various types of SynInL expressions. In all cases, we require that the index of the argument `Idx1` is PROLOG `==` to the index of the SynInL expression `Idx`.

`compile.pl` and `lexical.pl` define the semantic compilation predicates and lexical access predicates respectively. They are very closely related. The basic idea is that `compile_semantics` looks at every lexical entry and tries to compute a "key" on the basis of some property of the semantics of the lexical entry which will allow efficient secondary indexing for lexical access during generation. `u_cs_process0` determines what the appropriate key is. This information is then stored under the key in a term of the form `c(Canon, Sem, Key, Ref)`. `Canon` is a canonical form of the semantics of the lexical entry. (Currently it is identical to the semantics of the lexical entry.) `Sem` is the semantics of the lexical entry. `Key` is the database key that the lexical entry is stored under, typically something like `#Paris`. `Ref` is the database reference of the lexical entry. The keys under which these `c/4` terms are stored are all of the form `!Name` where Name is the key determined by `u_cs_process0`. For bookkeeping purposes this key is also stored in the database under the key `#sem`. This makes it possible to keep track of where the `c/4` terms are. The details of the heuristics that `u_cs_process0` uses to determine a key are very implementation and grammar dependent and highly likely to change. They are also of no theoretical interest so they won't be described

here. Further details can be found in the comments in the code.

`lexical.pl` defines `u_lexical` and `u_identity`. `u_identity` is a specialised version of `u_lexical` that assumes a lexical access key of `!identity`. `u_lexical` calls `u_lexical_key_type` with a sign `Sign` and returns a lexical access key `SemsKey`. It then calls `u_lexical0` with `Sign` and `SemsKey` to use `SemsKey` to find a lexical entry that will unify with `Sign`.

`u_lexical_key_type` can return three types of key: `hash(Key)` where `Key` is prefixed with `#`, `pling(Key)` where `Key` is prefixed with `!` and `template(Template,Tanker)` where `Template` is a template name and `Tanker` is an atom of an integer-suffixed proper name.

If the key is a `hash` type this means that the phonology of `sign` was instantiated and so we can use the key to efficiently find the correct lexical entry.

If the key is a `template` type then it means that `u_lexical0` has to create a sign using template `Template` since there is no corresponding lexical entry. Currently this is possible for templates `Tanker`, `Truck` and `Depot`. i.e., we can generate proper names like *truck27, tanker13* and *depot137.*

If the key is a `pling` type this means that the key was determined on the basis of the semantics. This is done by `u_key`. As with the code in `compile.pl` this predicate is volatile and theoretically uninteresting. We will not describe it further.

Chapter 4

French Grammar ; Parser and Generator

Thierry Guillotin,
Agnès Plainfossé
LdM

The French grammar underlying parsing and generation is integrated in the French Dialogue System, which is described in this chapter. The French Dialogue System has been built during and for the ACORD project. It is used both for query analysis and generation of answers. It includes :

- The French Grammar
- The Morphology
- The Parsing process
- The Generation process

4.1 Scientific Background and Development

After the feasibility study on the adaptation to French of GPSG grammars (cf. Deliverable T2.2), and as a result of the specification of the UCG grammar (Deliverable T2.1), we adopted this framework for defining the French grammar of the ACORD system. French interrrogative structures were specified in Deliverable T2.3 and its first version implemented in the French Dialogue Parser (Deliverable T2.7). The system has been extended as regards anaphora resolution (cf. Deliverable T1.7'a and Deliverable T1.7'b). The final version embeds the common representation of plurals. Its architecture is presented in the next section.

4.1.1 Architecture

The following schema shows the architecture of the French Dialogue System and the flow of information from the sentence reader to the end of the process for analysis.

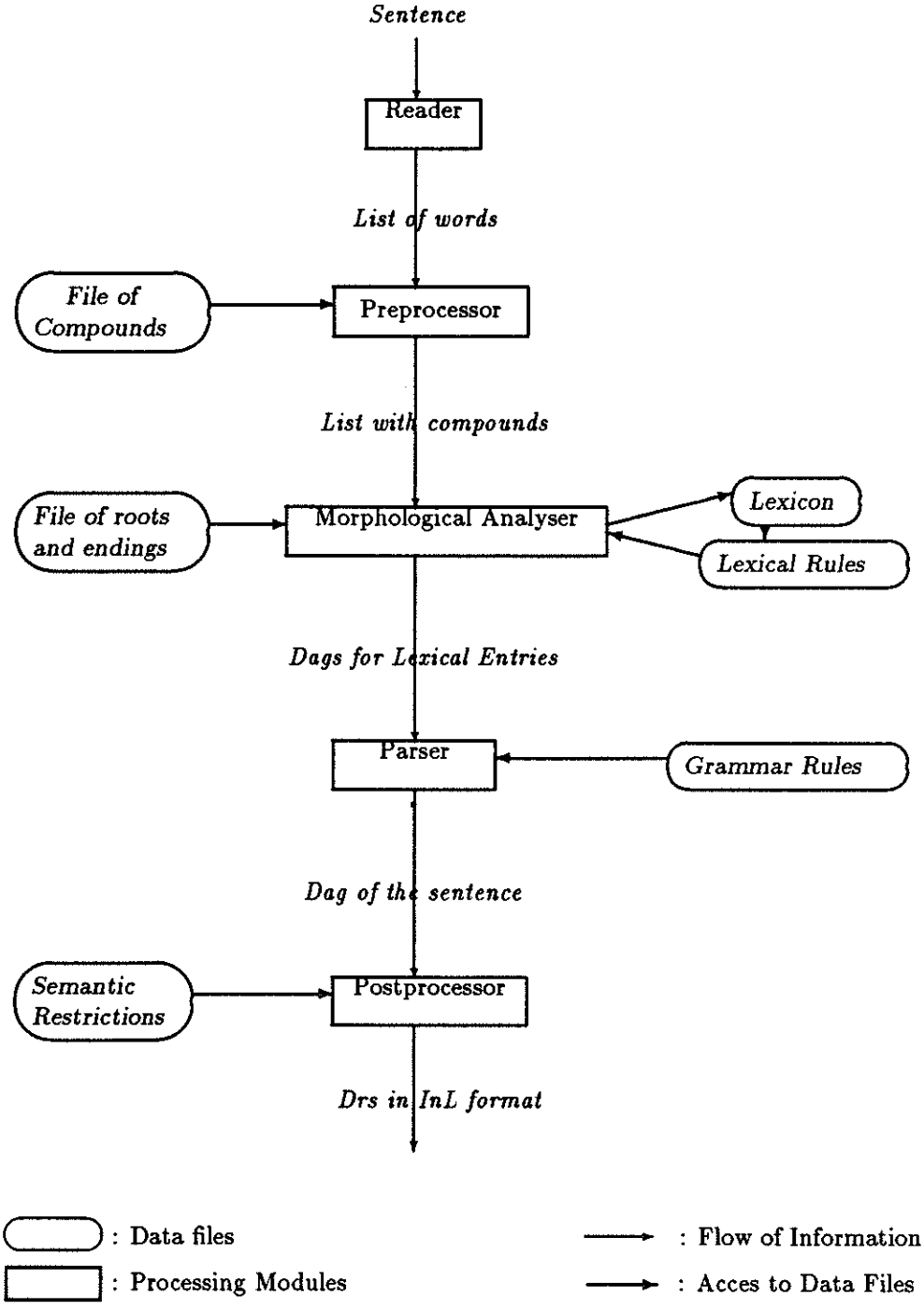


Figure 0.1: French Dialogue parser

4.1.2 Grammar Coverage

Considerable effort has been put into the description of interrogative structures in French. The general theory independent description given in Deliverable T2.3 This has been reformulated according to the UCG framework.

The current grammar covers *yes/no questions* and *wh questions.* Both types of questions require the treatment of movement phenomena, i.e. verb/subject inversion or *wh movement.*

Other stuctures that can be parsed cover :

Nominal and adjectival phrases
- nouns
- personal pronouns
- determiners
- intersective adjectives
- interrogative nominal structures

Prepositions
- case marking prepositions
- verbal prepositions
- including interrogative prepositional structures

Verbal base forms
- with Noun Phrase arguments
- with Prepositional Phrase arguments
- including interrogative verbal forms

Relative clauses
- NP argument of the verb
- Prepositional argument

Auxiliaries
- with passive participle
- with past participle

Adjuncts
- Noun modifiers
- Verb Phrase modifiers
- Sentence modifiers

4.2 Technical Description

4.2.1 Software and Hardware Requirements – Files

The grammar components need PIMPLE and PROLOG to be compiled into PROLOG terms. The French Dialogue System has the following directories/files structure :

PIMPLE files are located in `upimp` (together with the parsers and the translation to ACORD common representations) which is about 1600 Kbytes. Grammar files are in `gram` (about 200 Kbytes). Generation PROLOG files are in `generation` (150 Kbytes). The morphology part of the system is in `morpho` (100 Kbytes). A bunch of tests is in `tests`. These can be run (after loading `upimp/batch`) by calling `fdp_test_file(input_file, output_file)` where `input_file` might have the form `tests/*.test` and `output_file` `user` or any file name you desire.

The grammar files are located in the directory `gram`.
The grammar rules file is : `proto.gram`

The definitions files :

the Sign exhaustive declaration : `declare.def`
the abbreviation file : `abbrev.def`
the file of the definitions files : `proto.def`

The definitions files (the templates) :

`aux.def`	`det.def`	`rel.def`
`cadj.def`	`disj.def`	`support_verb.def`
`clitic.def`	`prep.def`	`templ.def`
`conj.def`	`neg.def`	`verb.def`
`control_verb.def`	`nomi.def`	
`copule.def`	`noun.def`	
`copuleadj.def`	`np.def`	

The lexical rules : `lexical_rule.def`
The internal semantic features declarations : `sorts.def`

The lexicon files (using definitions templates from the `*.def` files) :

`adj.lex`	`disj.lex`	`pers.lex`
`aux.lex`	`ei.lex`	`ponct.lex`
`cadj.lex`	`lex.lex`	`poss.lex`
`clitic.lex`	`names.lex`	`prep.lex`
`conj.lex`	`neg.lex`	`proto.lex`
`control_verb.lex`	`nomi.lex`	`rel.lex`
`copule.lex`	`noun.lex`	`support_verb.lex`
`copuleadj.lex`	`np.lex`	`verb.lex`
`det.lex`	`part.lex`	

4.2.2 The Grammar Formalism

The main building blocks of the grammar are *grammar rules*, *lexical entries* and *lexical rules*.

As in any categorial grammar, lexical entries constitute the basis of the grammar. They are built from abstract generic *templates* and are specified in more detail in each specific lexical entry.

Considering the fact that French morphology is too rich to allow the storage of one lexical entry per inflected form, we chose to store only the basic form of the word and to introduce a morphological component into the system. Full lexical entries are in this way constructed dynamically during the parse.

A word of the sentence is analysed morphologically ; values for person, gender and number resulting from this analysis are used to instantiate the corresponding features in the basic lexical entry, the value for the tense feature is used to spawn the corresponding lexical rule which will modify the lexical entry obtained by the previous instantiations.

The way of combining two lexical entries is given by the grammar rules.

Semantic representations of parsed sentences are given in InL format, an indexed language allowing the implementation of Discourse Representation Theory as described by Kamp 81. In this framework we elaborated a sort system for specification of the semantic sort of a lexical entry. Part of this information contributes to the restriction of possible references for pronouns.

In UCG the basic object the processes will work on are signs (structures which embody all linguistic knowledge). Lexical entries are signs; lexical rules operate transformations on signs, and grammar rules produce/derive new signs from signs that they combine.

UCG sign model for the French Grammar :

```
Sign -> Category:Mtrans:Semantics:Order:Occurrence:Phonology

Category -> Head^[Feat,Clitic_order,Agree]
Category -> Category/Sign
        Head -> sent | noun | np | pp
        Feat -> (fin | inf | pas | psp | cfin) | (subj | obj | a)
        Clitic_order -> v | le | lui | en | y | te | me | se
        Mtrans -> (Wh,Neg)
                Wh -> que | nque | wh | whs | inv2
                Neg -> nneg | neg | neg1 | neg2
        Agree -> (Ge:Nb:Pe)
                Ge -> masc | fem
                Nb -> sg | pl
                Pe -> 1 | 2 | 3

Semantics -> Predicate(Index,Arg1, ... ,Argn)
        Index -> 1:loc:movable:mass:plural:human:sex:Kb_Class:Identifier
        Index -> 0:movement:_:state:progress:phase:telic:Kb_Class:Identifier
                loc | movable | ... | telic -> 1 | 0
                Kb_Class -> atomic value1
                Identifier -> Variable | atomic value
        Predicate -> atomic value
        Argn -> Index | Semantics

Order -> pre | post | opt | jump
Occurrence -> [] | Index:[Position,Agree,Gram_function,Class,Access]
        Position -> integer
        Gram_function -> subj | obj | pobj | adj
        Class -> card | def | indef | name | demo | pers
                | french (for french ucg resolved possessives)
                | cla (for french clitic adjunct)
        Access -> [] | Occurrence | occ(Occurrence)
Phono -> Logy:Aux:Mode:Roots:Agree:Tense
Aux -> avoir | etre
Mode -> ''some morphological class identified by a prototypical verb name''
Roots -> ''the roots needed by the prototypical morphological class to apply''
Agree -> ''the Morphological analyser will give the precise actual values''
Tense -> ''value for tense delivered by the morphological analyser''
```

So a sign of the UCG French System looks like :

```
 Head^[feat,Clitic_order,Agree]/Sign1/.../SignN
:(Wh,Neg)
:(Index:Predicate:[Arg1,...,ArgN])
:Occurrence
:Order
:(Phono:Logy:Aux:Mode:Roots:Agree:Tense)
```

Lexical entries. Example :

```
camion

 noun^[nil,nil,(masc:sg:3)]
:_
:(X:truck:[])
:(X:occ:[0,(masc:sg:3),_,_,_])
:Order
:(camion:[mur]:[camion]:(masc:sg:3):nm)

conduit

 sent^[fin,_,(Ge:sg:3)]
        /(np^[subj,_,(Ge:sg:3)]:_:X:(_:occ:[_,_,subj,_,Occ]):pre:_)
        /(np^[obj,_,_]:_:Y:(_:occ:[_,_,obj,_,(_:occ:[_,_,subj,_,Occ])]):_:_)
:(nneg,nque)
:(E:drive:[X,Y])
:Occurrence
:Order
:(conduit:avoir:[cuire]:[condui,conduis]:(Ge:sg:3):ip)
```

Lexical rules. Example :

```
Passive :

 sent^[_,Cl,A]/(np^[subj,_,A]:_:X:OccX:pre:Cl1)
              /(np^[obj,_,_]:_:Y:OccY:post:_)
:MT
:Semantics
:Occ
:O
:P

-->
```

```
 sent^[pas,Cl,A]/(np^[subj,_,A]:_:Y:OccX:pre:Cl1)
                 /(pp^[par|_]:_:X:(_:occ:[_,_,pobj,_,OccX]):post or opt:_)
:MT
:Semantics
:Occ
:O
:P
```

Grammar rules

Binary Rules :

```
f_apply =>
        C:MT:S:Occ:O:(['>apply',W1,W2]:nil:nil)
        -->
        (C/ #1) :MT:S:Occ:O:(W1:_:_)
        ,
        1#(C1:MT1:S1:Occ1:pre:(W2:_:_)).

b_apply =>
        C:MT:S:Occ:O:(['<apply',W1,W2]:nil:nil)
        -->
        1#(C1:MT1:S1:Occ1:post:(W1:_:_))
        ,
        (C/ #1) :MT:S:Occ:O:(W2:_:_).
```

Apart from the Binary Functional Application (Forward and Backward) rule, a *Forward "Gap" Rule* has been designed to handle "movement" phenomena.

```
f_gap  =>
        C/(#1:NW:SO:OccO:pre:(#2)):MT:S:Occ:O:(['>gap',W1,W2]:nil:nil)
        -->
        C/(C1:MT1:S1:Occ1:pre:W2):MT:S:Occ:O:(W1:_:_)
        ,
        C1/(1#((np or pp)^[~subj|_]):NW:SO:OccO:_:2#(~lex)):MT1:S1
         :Occ1:pre:(W2:_:_).
```

Several *Unary Rules* handle order of arguments (jump), optionality of arguments (delopt), addition of adjunct arguments on verbs (ppv), nouns (ppn) and sentence fragments (pps).

```
jump  =>
        C/(#2:M2:S2:Occ2:post:P2)/(C3:M3:S3:Occ3:O3:P3)
        :MT:S:Occ:O:([jump,W]:P:M)
        -->
        C/(C3:M3:S3:Occ3:O3:P3)/(2#((np or pp)^_):M2:S2:Occ2:jump:P2)
        :MT:S:Occ:O:(W:P:M).

delopt =>
        C:MT:S:Occ:O:([delopt,W]:P:M)
        -->
        C/((np or pp)^_:_:(@sem_opt):_:opt:_):MT:S:Occ:O:(W:P:M).

ppv =>
        2#(sent^[(fin or inf or imp)|_]
               /(np^[subj|_]:_:_:Occs:pre:_))
               /(pp^[adj|_]:_:I:(_:occ:[_,_,adj,_,Occs]):post:_)
        :(#3):1#(I:_:_):Occ:O:([ppv,W]:P:Mo)
        -->
        #2:3#(_,_):(#1):Occ:O:(W:P:Mo).

ppv =>
        2#(sent^[(fin or inf or imp)|_]
               /(np^[subj|_]:_:_:Occs:pre:_)
               /((np or pp)^_:_:_:_:_:_) )
               /(pp^[adj|_]:_:I:(_:occ:[_,_,adj,_,Occs]):post:_)
        :(#3):1#(I:_:_):Occ:O:([ppv,W]:P:Mo)
        -->
        #2:3#(_,_):(#1):Occ:O:(W:P:Mo).

ppn =>
        2#(noun^[_,(nil or adj),_])/(pp^[adj|_]:_:I
           :(_:occ:[_,_,adj,_,Occ]):post:lex)
        :Mt:1#(I:_:_):Occ:O:([ppn,W]:P:M)
        -->
        #2:Mt:(#1):Occ:O:(W:P:M).

pps =>
        2#(sent^[fin,_,_])
               /(pp^[adj|_]:_:(T:O:N:M:A:B:C:KB:V)
                 :(_:occ:[_,_,adj,_,_]):pre:lex)
               /(lex:_:_:_:post:(',':_:_))
        :NW:1#((T:L:N:M:A:B:C:KB:V):_:_):Occ:O:([pps,W]:P:Mo)
        -->
        #2:NW:(#1):Occ:O:(W:P:Mo).
```

4.2.3 Extensions to PIMPLE

The PIMPLE environment has been modified in several ways to support the French Grammar design. The main extension concerns the constraints handling. Originally, PIMPLE was able to handle negative and disjunctive constraints (see above **1.1**). We have introduced a new facility which concerns the *co-relational constraints.*

This facility allows to express the relations between different fields of the *sign* as the following example suggests, which co-relates the semantics of the noun to its morphological agreement :

```
[np^[_,_,1#(masc:Number:3)] / (noun^[_,_,#1]:_:S:_:_:_)
 :_
 :Semantics
 :Occ
 :Order
 :(_:_:_:#1:_)
 { [Number] ==> [Semantics]
   case
   [sg] ==> [@singular(S)]
   or
   [pl] ==> [@plural(S)]
 }
]
```

These complex constraints basically allow to express disjunction over sets of variables. A set of variables S_2 ($[Var'_1, Var'_2, \ldots, Var'_m]$) is dependent on another set of variables S_1 ($[Var_1, Var_2, \ldots, Var_n]$): following the values V_1 that S_1 will admit, S_2 will take *co-relationally* V_2 as values.

They obey the following syntax :

$[Var_1, Var_2, \ldots, Var_n] \Longrightarrow [Var'_1, Var'_2, \ldots, Var'_m]$
case
$[x_1, x_2, \ldots, x_n] \Longrightarrow [x'_1, x'_2, \ldots, x'_m]$
or
$[y_1, y_2, \ldots, y_n] \Longrightarrow [y'_1, y'_2, \ldots, y'_m]$
or
...

where Var_i is a PROLOG variable, and x_i either a PROLOG term or a negative or disjunctive constraint which Var_i handles.

They are treated in the following way :

(1) if S_1 takes its full set of values in the disjunction $V_1, \ldots, V_z$ then S_2 will be assigned the related set of values from the disjunction $V'_1, \ldots, V'_z$

(2) if it doesn't then the whole co-relational constraints expression will be reduced logically following the subset of values S_1 has taken and the remaining disjunction will be calculated.
Example :
On the constraint :

$[V_1, V_2] \Longrightarrow [V_3]$ case $[a, b] \Longrightarrow [c]$ or $[a, e] \Longrightarrow [f]$ or $[g, h] \Longrightarrow [i]$

if V_1 only gets instanciated (to a) then the expression will be reduced and will become :

$[V_2] \Longrightarrow [V_3]$ case $[b] \Longrightarrow [c]$ or $[e] \Longrightarrow [f]$

The predicate which handles the contraints in PIMPLE is `p_check_constraints` (located in `upimp/pdag.pl`).

The co-relational constraints are processed in the opposite way ($\Longleftarrow$) for the purpose of generation by the `gen_check_constraints` predicate which can be found in `generation/nglex.pl`.

4.2.4 The Morphology Processes

Two different processes operate. The first one treats the morphological information (roots, mode) from all the lexical signs to store the roots on the roots trees and the endings (which are general to a mode) on the endings trees. The modes are declared distinctly from the lexicon (`morfo/fdp_mode_conj.pl`).

Example :

```
noun^_:_:_:_:_:(carafe:_:[balle]:[caraf]:_:_)

noun^_:_:_:_:_:(carafon:_:[mur]:[carafon]:_:_)
```

The first declaration from the lexicon behaves as "balle" with root "caraf", the second one as "mur" with "carafon" as root. "mur" is the prototype for morphogically masculine nouns allowing singular and plural derivations, "balle" is the same kind of prototype

for feminine; the two require only one unique root to be declared. These entries will generate the following tree for roots :

```
[c,[a,[r,[a,[f,[[],(carafe, balle),
               [o,[n,[[],(carafon,mur)]]]]]]]]]
```

This allows to store the root lexical entries and derive by morphological analysis the effective dynamic lexical entries.

The second process is the one which will decompose the actual word used in a sentence into a root and an ending which are compatible by searching in the roots and endings trees. This process will deliver the **agreement**, **tense** and root label to access the lexicon. The top level call for that process is :

```
morph_an(InList, Word, RootWord, (Tense,Pers,Gender,Number,Aux), OutList)

W = carafes
Rw = carafe
Mf = (nf,3,fem,sg,nil)
Ol = []
```

So a root lexical entry as the following :

```
noun^[_,_,1#(masc:Number:3)]
:_
:[X:and:[Sem,(X:truck:[])]]
:OccX
:Order
:(camion:_:[mur]:[camion]:#1:_)
,
{ [Number] ==> [Sem] case
  [sg]     ==> [(X:singular:[])] or
  [pl]     ==> [(X:plural:[])] }
```

will, after the morphological analysis, become :

```
noun^[_,_,(masc:sg:3)]
:_
:[X:singular:[],(X:truck:[])]
:OccX
:Order
:(camion:_:[mur]:[camion]:(masc:sg:3):_)
```

4.2.5 The Parser

Two different parsers can run using the Grammar Data (morphology, lexical entries, lexical rules and grammar rules). The first one, which is very interesting for debugging purposes because it can give diagnosis on failures that have occurred in the course of the parsing process, is the chart parser (`upimp/chart_parser.pl`); the other one which establishes fewer hypotheses from the data and then uses less storage capacities is the shift reduce one (`upimp/shift_reduce_parser.pl`). Both parsers include the morphological analysis and its link to the lexical rules and the lexical access (`upimp/plex.pl`).

4.2.6 The Generator

French Generator is the label for the French specific modules for generation. French generation is not restricted to these modules, especially the second version which includes a substantial planning component.

The French Generator uses the same grammar definitions (morphology, lexical entries, lexical rules and grammar rules) as the parser. It can be viewed as an inverse parsing since it shares the same resources.

There have been two approaches to the generation process. The first one is more rudimentary : it takes as input a resolved InL expression, the second one which involves a planning component, takes a SynInL expression as input (see above **2.1**).

Both versions have basically the same architecture at the French UCG level. Their main differences at this level reside in handling InL and SynInL respectively, and in terms of a different lexical access.

The generation modules all reside in the subdirectory `/generation`. The file names corresponding to the second version have been prefixed by the letter 'n' standing for 'new'. The modules are :

version	Name	Function
(1)	gsem.pl	semantics compilation
(2)	ngsem.pl	
(1)	glex.pl	lexical access
(2)	nglex.pl	
(1)	gen.pl	top level functions
(2)	ngen.pl	
(2)	ngen1.pl	
(1) (2)	gmorph.pl	morphological processing
(1)	ginter.pl	generation interface
(2)	nginter.pl	

Lexicon compilation and lexical access for generation The lexical access in generation is mainly performed using semantical information. Since the lexicon was primarily designed for parsing purposes, this access can be very long and inefficient. To perform a faster lexicon access in generation, a compilation of the lexicon has been implemented. Lexical compilation enables a fast access to lexical entries using semantics as a search key as well as a syntactic key for the second version generator. The compilation is run once to create an intermediary search table. The lexical compilation functions can be found in **gsem.pl (ngsem.pl)** and **glex.pl (nglex.pl).**

The compilation of the whole lexicon is called by the top level function :

(1), (2) cs.

The compilation of a particular lexical entry is performed using :

(1) cs('Name') ex : cs('Camion')
(2) cs(File-Name,'Name') ex : cs('gram/noun.lex','trouver').

where :

Name is the key word of the lexical entry such as "camion"
File-Name is the grammar file name where this lexical entry is defined

The cs function creates an intermediary table made up of the following records :

(1) record(Sem-Key,sk(Canon-Semantics,Occ-Semantics,Semantics,Word-Key,Poin ter)
(2) record(Cat-Key,sk(Sem-Key,Occ-Semantics, Semantics,Occ,Word-Key,Pointer)

where :

Cat-Key	:	syntactical key
Sem-Key	:	semantical key
Canon-Semantics	:	semantics under its canonical form
Occ-Semantics	:	semantics with occurrence information
Semantics	:	semantics with occurrence information removed
Occ	:	occurrence information
Pointer	:	pointer to the corresponding lexicon record

Cat-Key provides the syntactic nature of the lexical entry and is defined using the File-Name where the lexical entry is defined within the grammar. Sem-Key is an atomic key compiled from the semantics by the u-key function. When the semantics of a sign is variable (identity semantics), u-key fails and the value 'identity' is assigned to Sem-Key.

The lexical accesses are then respectively performed using :

(1)-a acces-lex(Sign:C)
(1)-b acces-lex-identity(Sign:C)

(2) acces-lex(Cat-Key,Sign:C,Occur)

Acces-lex compiles Sem-Key using the semantics provided in Sign, then it accesses the intermediary table using this Sem-Key (Cat-Key and Sem-Key). The intermediary table provides the lexical entry which corresponds to the input sign. In the first version, there is a distinction between regular and identity lexical access.

The morphology process for UCG generation The morphological main tasks to be handled in generation are first to be able to reconstruct the final word from a root and morphological information . This process uses the same tables as the parser and the corresponding functions are defined in **gmorph.pl (ngmorph.pl)** with the top function `morph_gen`.

Another task consists in helping the process of dynamic building of lexical entries for proper names (*Pierre, Camion1* ...).

The algorithms for UCG generation The algorithms for UCG generation are common to French and English UCG generation and are fully described in the Deliverable T2.10 under the chapter 2.3 for the first version and 3.4 for the second version.

The first version is implemented in **gen.pl**

As far as the second version is concerned, there are two different implementations. The one corresponding to the general algorithm described in the deliverable can be found in **ngen1.pl**. A faster but less general algorithm is implemented in **ngen.pl**. This last algorithm is very close to the SynInL specification within the coverage in ACORD. It holds restrictive expectations on the sentence to be produced. Nevertheless, it is very efficient and fast.

4.2.7 User Interfaces

The standard menus have the following options.

The *Start Up* menu

```
Type

abort                to abort
b                    to enter a break
```

```
browse              to examine previous parse
file F              to send output to file F
reset               to resume sending output to the screen
n                   to end prolog debugging
redo                to reparse previous sentence
var                 to examine the global variables
lg                  to (re)load the grammar
ld                  to (re)load the definitions
ll                  to (re)load the lexicon
ld File             to (re)load the definition file File.def
ll File             to (re)load the lexicon file File.lex
l File              to (re)load the files File.lex & File.def
x                   to exhaustively trace (prolog debugging)
q                   to return to prolog
quit                to return to prolog
halt                to return to Unix
String              to parse String
```

Help for all commands in a menu is available on-line.

The French Dialogue System menu

```
Type

a                   to abort
b                   to enter a break
c                   to continue (return to parse level)
d                   to show DAG structure of current edge(s)
gen                 to generate sentences from the current InL(s)
file F              to save whole chart in file F
n                   to end prolog fdp_debugging
p                   to print the vertex list
qp                  to print the current edges quickly
r                   to run back end procedure on output
                        (towards ACORD representation +
                         anaphora resolution)
antec               to see antecedents (anaphora resolution)
curr                to see occurrences  (anaphora resolution)
clean               to clean dialogue history Data Base
                        (anaphora resolution)
inl                 Drs in Inl output
inlocc              Drs in Inl_Occ output
syninl              from InL to SynInL format (format for Generation)
s                   to show statistics on current parse
t                   to show parse tree(s) for the whole string
```

```
t M N                to show parse tree(s) for the string between
                        M and N
v                    to see edges
v N                  to see edges at vertex N
top_level T          to set top_level_goal_is_template to T
rule_dag R           to show the rule R
word_dag W           to show the DAG structure of word W
template_dag T       to show the DAG structure of template T
lexical_rule_dag LR  to show the lexical rule LR
spy P                to spy P
nospy P              to unspy P
spypoints            to show spy points
x                    to exhaustively trace (prolog fdp_debugging)
```

The *Start-up* file System parameters, such as the name of a grammar, the location of user-customization files and variables which control the behaviour of parsers, may be given in a start-up file. The format of this is as follows.

```
fdp_set(grammar_name, proto).
fdp_set(parser_directory, '../upimp/').
fdp_set(fdp_parser, shift_reduce_parser).
fdp_set(format_for_ppinl, short_format).

fdp_set(grammar_directory, '../gram/').
fdp_set(morfo_directory, '../morfo/').
fdp_set(path_for_lex_symbol, phono:logy).
fdp_set(debugging_level, 2).
fdp_set(back_end, fdp_dm_nl_resolve).
fdp_set(path_for_position_number, phono:position).
```

4.2.8 Relations to Other ACORD Components

Interfaces with other parts of the ACORD system In order to communicate with the other components, four predicates are defined in the file `sys_nl_fre`. These are `nl_known_word/3`, `nl_parse_sentence/3`, `tg_generate_sentence/4` and `tg_new_generate_sentence/3`.

`nl_known_word(french, List, Difference)` succeeds if, in the language `Language`, the list `List` begins with a sequence of tokens that represent a valid lexical item. `Difference` is the difference of `List` with respect to those tokens.

`nl_known_word(Sentence, french, Result)` succeeds if `Sentence` is a grammatical sentence in `Language`, in which case `Result` is instantiated to the corresponding interface representation, including the sentence's translation into InL.

`tg_generate_sentence(french, Term, InL, Sentence)` succeeds if InL corresponds to the semantics of an assertion. In this case, Sentence takes the value of the corresponding French sentence in the form of a list of token words produced by the first version of the French Generator. `Term` is the term the initial question was asking about. It enables some canonicity checking before generation can start (**ginter.pl**).

`tg_new_generate_sentence(french, SynAnswer,Sentence)` succeeds if `SynAnswer` corresponds to the semantics of an assertion. In this case, `Sentence` takes the value of the corresponding French sentence in the form of a list of token words produced by the second version of the French Generator. `SynAnswer` can possibly be prefixed by markers to which correspond a canned French text such as *non, oui et meme.*

The French Dialogue System is included in the Natural Language Process of the system together with the English and the German ones. It takes a list of words as input and delivers as output an InL formula, including the Occurrence information for each noun phrase to be handled by the anaphora resolution component.

Mapping between the French Dialogue System and ACORD Representations Given the different styles of representation used by the ACORD system as a whole and by the French grammar, in respect to semantic representation, we provide a mapping between the two representations. The mapping is relative simple, as there is an embedding of the ACORD sort system into the French sort system, and a simple isomorphism between the semantic representations. These are computed by the routines called by `fdp_dm_translation`.

4.3 Potential of Developments

Use in other situations The Grammar in itself is portable to other domains in the context of dialogue situations together with the RESOLVER (see **7**).

Implementations in other languages A subset of the French UCG Grammar has been implemented using the Object Oriented language Lore.

Use with other components of the system The French system can be loaded together with the Generation Planner in order to test the path from the analysis of a sentence and its generation. The analysis part of the system can be loaded with the RESOLVER (and the KB frame system, see **11**) and allows to include anaphora resolution, local consistency checking and adjunct desambiguations as post-processes to the parsing process.

Chapter 5

German Grammar ; Parser and Generator

Andreas Eisele,
Walter Kasper,
Dieter Kohl,
Klaus Netter
IMS

The Stuttgart LFG system for the development of LFG grammars is described in this chapter. The scientific background and history of the component is presented first and then comes a discussion on its implementation and use.

5.1 Scientific Background and Development

5.1.1 The Grammar Formalism

Both the formalism and the linguistic theory of Lexical Functional Grammar (LFG) were originally developed as an alternative to traditional Transformational Grammars. (Bresnan 82, specifically Kaplan and Bresnan 82).

On the *linguistic* level the two major assumptions were :

- that given the variety of word order phenomena in different types of languages grammatical relations should not be encoded exclusively in terms of tree configurations but rather in terms of *grammatical functions* as primitive notions of the

description language.

- that many variations in the surface structure of related expressions, such as passives, raising etc., should not be captured by structural transformations operating on tree structures but rather by *lexical processes*, defining the structural environments into which a lexical item can be inserted.

The linguistic theory of LFG is matched by a *grammar formalism* which provides a rich and expressive description language. The linguistic information associated with a natural language expression does not necessarily have to be reduced to a single data type, namely phrase structure trees, but can be represented by different types of structure tailored to suit the individual types of information.

Phrase structure trees of an X-bar oriented form, so-called constituent or *c-structures* serve to represent the phrasal configuration and elementary structure of parts of speech in terms of precedence and dominance relations obeying a single mother condition.

A second data type are attribute value structures, so-called functional or *f-structures*, with atomic and complex valued features, including set valued features and attributes with shared values. *f*-structures allow to abstract from linear surface order and encode information such as grammatical relations holding between parts of speech, underlying predicate argument structures, coreference between complements, agreement or government relations, etc.

c-structures are being described by ordinary context-free phrase structure rules, *f*-structures by sets of equations annotated to rhs symbols of PS-rules. The annotations to PS-symbols yield descriptions of *f*-structures which not only contain structure building equations but also constraint equations, enforcing the presence or absence of specific attribute value pairs, which a given *f*-structure has to satisfy.

One of the major modifications to LFG that occurred since 1982 concerns the treatment of non-local dependencies (NLDs). Whereas in the original version NLDs were basically defined over c-structures while simultaneously involving a filtering effect of *f*-structures, the use of the so-called *functional uncertainty* mechanism allows to define non-local dependencies by means of *f*-structure paths using regular expressions over the functional attributes. (Kaplan and Maxwell 88)

5.1.2 The Parser

The parser of the LFG system consists of two parts: The *parser compiler*, that takes a grammar and a lexicon given in the LFG formalism and transforms it into the internal representation used by the rest of the system, and the proper *parser*, which analyses sentences and phrases by constructing c-structures and *f*-structures for them according to a grammar and a lexicon that have been compiled before. Both parts use a set of routines for the unification of partially specified *f*-structures that allow for the efficient representation of disjunctive information and provide extensions to the standard

methods for (e.g. PATR-style) feature-unification which are part of the LFG-formalism.

The formalism that can be used has been defined for earlier versions of the system (see Deliverable T1.4 or Eisele and Dörre 86) and has not changed very much since then. It provides the annotation of a context free phrase structure grammar and a lexicon with descriptions concerning the feature structures of the constituents. Such descriptions include equations between different paths or between paths and atomic values, where the latter can also be negated or interpreted as constraining equations, and where the paths may also involve indirectly specified features. The descriptions can express set inclusion and positive and negative existential constraints, and can be combined conjunctively or disjunctively. A detailed description of the formalism with examples is given in the next section of this document.

While reading a given LFG-grammar, the parser compiler transforms it into an equivalent one without kleene-star and optional constituents; it compiles the annotations of the grammar rules into partial f-structures containing the same information; and finally, it constructs an SLR-parser table (see Aho et al. 86) from the context free-skeleton of the (modified) grammar.

The parser includes a table-driven shift-reduce parser for context free grammars that collects all possible c-structures for a given input sentence in a parse forest and uses a graph-structured stack as data structure to represent different branches of the analysis. After the complete parse-forest has been found, a second component traverses it and unifies the partial f-structures of the involved constituents in order to find the f-structure(s) of the overall input. The main architecture of the system has been described in Eisele and Schimpf 87.

In the implementation of f-structure-unification, the techniques described in Eisele 87 and Eisele and Dorre 88 have been used. Due to these techniques, the system can process grammars containing a high amount of disjunctive specifications without having to multiply out all possibilities.

5.1.3 Construction of InLs from f-structures

For the German grammar, using the LFG formalism we chose to develop the mapping from f-structures to semantic representation as a separate module, since the LFG formalism itself was not well suited to express it, especially with respect to scope. This also had the advantage that syntax and semantics could not only be developed independently, but also more easily adapted to the developing requirements of ACORD. Reyle and Frey 85 developed the basis for an algorithm for such a mapping using DRT as semantic framework which had also been advocated in Deliverable T1.2 and was presupposed in Deliverable T1.3. This algorithm exploited the order freeness of f-structures for deriving scope ambiguities, and uses functional application as basic mechanism for building the semantics. After, as a variant of DRT, InL was chosen as semantic representation language, this algorithm was adapted and re-implemented for InL. At the same time it became necessary to develop the basic algorithm further, since in its original form

it was only suited to deal with simple sentences consisting of verbs and their subcategorizing functions, and also did not account for the needs of anaphora resolution. So the first implementation was accompanied by an own facility for doing anaphora resolution (Kasper and Reinhardt 88, Deliverable T2.1) which became superfluous and was dropped when ACORD decided to use a central RESOLVER. Instead the algorithm had to be adapted to provide the new interface structures needed for the RESOLVER.

In order to cover all of the syntactic fragment of German and to meet the requirements of semantic representations in ACORD the algorithm had to be augmented in other respects as well, especially with respect to the control of scope by use of surface structural information, restricting the number of possible derivations by stronger typing and enhancing the control of the flow of information by imposing rules for special forms of f-structures. Also, the treatment of plurals in ACORD made it necessary to let the semantic representation be induced not just from a mapping of atomic values of features to semantic values, but to allow for mappings from partial f-structures.

The module has been written directly in PROLOG, and the code in many places relies on the special format of InL. In the last years some of this dependency has been reduced, but the adaptation to other forms of semantic representation would still require a lot of rewriting, though the structure of the module is fairly general. The portability is also reduced, because a specific form of internal representation of f-structures is presupposed, and some of its properties are also special to the ACORD grammar. Considerations concerning a more high level specification for the module have not yet reached the stage of implementation.

5.1.4 Construction of f-structures

To generate sentences in an LFG framework it is the natural choice to use a generator, which generates phrases from f-structures. Thus it has been necessary from the time we used generation in ACORD, to have a separate module for the construction of f-structures from a semantic representation.

From the beginning, the basic assumption for the f-structure generator was that the input structure should explicitly or implicitly provide all information necessary to make the various distinctions between

- sentence types, e.g. relative clauses vs. sentence coordination etc.
- choice of a name, a pronoun or an NP description
- where to negate, if there is a negation
- when to use quantifiers or if-then constructions
- which adjuncts are to be chosen, and with which part of the phrase they are associated

Until the beginning of 1989 the *f*-structure generator had generated from InLs. The experiences made with this generator had influenced the development of the planning component (see Deliverable T2.10), since the *f*-structure generator used internally an intermediate structure, which has much in common with SynInL.

With the choice of SynInL in the new overall generation concept the *f*-structure generator had been rewritten to work on SynInL. Much code of the former *f*-structure generator could be used or needed only slightly modifications, which has been possible due to a modular conception of the original *f*-structure generator.

5.1.5 The Generator

The generator in the narrower sense is the phrase generator, which produces phrases of some (given) syntactic category from *f*-structures. The phrase generator is based on the derivation algorithm from Jürgen Wedekind as described in Wedekind 86. A first version had been available in 1987 and was described in Dorre and Momma 87. The phrase generator works with a breadth-first top-down approach and generates by working on the c-structure and the *f*-structure in parallel, allowing the introduction of valid atomic valued features.

The needs of ACORD and other projects, where the phrase generator is used, lead to the development of a special compiler for the needs of generation. This compiler takes an LFG written in the format used in Stuttgart and makes some rearrangements of the grammatical rules, lexical entries and the internal order of the *f*-descriptions, to allow the generator to make the access to the single parts in an efficient order and to avoid some problems, which arise from the PROLOG treatment of negation in some special cases. The compiler also collects information about possible attribute value pairs, which are needed during generation as well, as for the debugging of the grammar.

The current version of the generator is well tested by now, and can be used not only in C-Prolog but also in other PROLOGs using the Edinburgh syntax. The generator can be loaded and run in QUINTUS PROLOG. Due to some incompatibilities in the compiler declarations, loading the generator into SICSTUS PROLOG causes some problems if one wants to use a makefile, but once loaded the phrase generator will work. Using a special patch file[1] the generator can be used also in ARITY PROLOG.

[1] Not provided with the ACORD version.

5.2 Technical Description

5.2.1 Software Description

All parts of the NL system from Stuttgart reside in subdirectories under $ACORD_STUT. We will describe the grammar as well as common submodules, which are used in other modules of the Stuttgart system, besides the software as mentioned above.

Directory hierarchy:

fina/	treatment of finite automatons
generation/	the phrase generator and the pretty printer
generation/synfsgen/	the *f*-structure generator on the basis of SynInLs
generation/synfsgen/unify	the unification program for the *f*-structure generator.
gram/	the German ACORD grammar
meaning/	the semantic constructor
tomita/compile/	the compiler for the tomita parser
tomita/parse/	the tomita parser

There are also the subdirectories for the PLANNER and the RESOLVER since both components were mainly developed in Stuttgart. These components are described in **2** and **7** respectively.

5.2.2 Software and Hardware Requirements

Except for the grammar, the software is written in PROLOG in the Edinburgh syntax. Most of the PROLOG code runs under QUINTUS PROLOG, SICSTUS PROLOG and ARITY PROLOG without changes, making use of different filename extensions in these PROLOG versions to load patch files, which are necessary to overcome incompatibilities between these PROLOG dialects.

To load the complete NL system into prolog+ you have to call it with at least

```
prolog -h2048 -a324 -g1024 -l512 -x32+
```

A saved state of a standalone system with the ACORD grammar takes about 1.7MB. For excessive pretty printing you may need to increase the local stack to `-l1024`.

Except for the loadfiles

sys_nl_ger.cpr
sys_tg_ger.cpr

where the first one is needed to load the complete NL part from Stuttgart into the ACORD system, no other file from Stuttgart makes any use of predicates which are not defined in one of the files which make the Stuttgart NL component.

The NL system including PLANNER and RESOLVER directories without a saved state of a standalone system needs about 1.7MB disk space.

5.2.3 The Grammar

All files which describe the grammar for German reside under $ACORD_STUT/gram. This directory contains also the dump files of the grammar as needed by the parser.

The grammar is written in the LFG format as it is used in Stuttgart. This format allows to describe lexical entries by a form of regular expressions, which has much in common with string equations. The compiler for the Tomita parser contains a special reader, to read such grammar files. This reader is also used by the compiler of the phrase generator.

As mentioned above, in the original form of LFG two levels of representations were assumed, c-structure and f-structure.

c-structures are described by standard context free rules, including optionality of symbols and other abbreviatory devices. The right hand side symbols as well as terminal symbols (the lexical items) are annotated with so-called *functional rule schemata* which define the correspondences between c-structure and f-structure nodes.

$$\begin{array}{lccc} S & \rightarrow & NP & VP \\ & & (\uparrow \text{SUBJ}) = \downarrow & \uparrow = \downarrow \end{array}$$

$$\begin{array}{lll} \text{repariert: V,} & (\uparrow \text{PRED}) & = \text{'repair} < (\uparrow \text{SUBJ}), (\uparrow \text{OBJ}) >' \\ & (\uparrow \text{SUBJ NUM}) & = \text{sg} \end{array}$$

The functional rule schemata contain metavariables ($\uparrow$ and $\downarrow$) which are instantiated with indices assigned to the respective mother node and the annotated node itself in the tree structure. These indices represent the arguments of a function ϕ into f-structure nodes, which are designated by terms of the form $\phi(i)$. [2] The f-structures can thus be interpreted as the minimal model of the set of equations which is obtained from the annotations by uniform and unique substitution of the designators by variables.

[2] As Kaplan 87 points out these rule schemata are just convenient abbreviations for a more complex notation making use of a mother function M defined for each node of the c-structure (except the root node). If we assume $\star$ to represent a c-structure node, then $\mathcal{M}(\star)$ defines its mother node and the metavairables can be rewritten as $\phi(\mathcal{M}(\star))$ for $\uparrow$ and $\phi(\star)$ for $\downarrow$.

The syntax of functional rule schemata in its basic form distinguishes between *defining equations* and *constraint equations.*

Defining equations with the equality operator have a non-empty string of attributes on one side and a atomic value on the other side or equations with potentially empty strings of attributes on both sides.

$$\begin{array}{lcl} (\uparrow \text{SUBJ NUM}) & = & \text{sg} \\ (\uparrow \text{SUBJ}) & = & (\uparrow \text{XCOMP SUBJ}) \\ \uparrow & = & \downarrow \end{array}$$

A specific form of a defining equation is used for set-valued attributes. Set-valued attributes have as their value not one single f-structure, but sets of f-structures, each element of which corresponds to a different c-structure node. Equations with the set-operator can be used to assign the same function such as ADJunct, or MODifier to more than one constituent.

$$\begin{array}{lcccc} \text{N1} & \rightarrow & \text{AP} & & \text{N1} \\ & & \downarrow \in (\uparrow \text{MOD}) & & \uparrow = \downarrow \end{array}$$

Whereas defining equations serve to construct an f-structure, constraint equations can be used to require that an f-structure must have a specific form. Positive Constraints

$$! \ (\uparrow \textit{Attribute})$$

require an attribute to be assigned a value by some defining equation, negative constraints

$$\sim (\uparrow \textit{Attribute})$$

require an attribute not to be assigned any value by a defining equation. Positive or negative existential constraints demand an attribute to be assigned a specific value or not, respectively:

$$(\uparrow \textit{Attribute}) =\text{c } \textit{atom} \text{ or}$$
$$(\uparrow \textit{Attribute}) =/ \textit{atom.}$$

The indirect specification of an attribute is a useful abbreviatory device which comes in handy if the choice of a grammatical function heavily depends on specific lexical items.

Let us assume, for example, that Prepositional OBJects of verbs such as *depend, blame, judge* etc. are subcategorized for as complex grammatical functions (↑ ON OBJ), (↑ FOR OBJ) (↑ WITH OBJ).

Prepositions are then entered in the lexicon accordingly by either being semantically meaningful (i.e. have a PRED-value subcatogorizing for an OBJ) or as pure case markers carrying an atomic valued feature PCASE:

$$\begin{array}{lll} \text{on: P,} & \{/\ (\uparrow \text{PRED}) & = \text{'on} < (\uparrow \text{OBJ}) > \text{'} \\ & /\ (\uparrow \text{PCASE}) & = \text{on}\ /\} \end{array}$$

while the uniform rule for PPs looks like:

$$\begin{array}{lccc} \text{PP} & \rightarrow & \text{P} & \text{N} \\ & & \uparrow = \downarrow & (\uparrow \text{OBJ}) = \downarrow \end{array}$$

Instead of annotating the PP-symbol in higher constituents by a disjunction of

$$\begin{array}{rl} \{/(\uparrow \text{ON OBJ}) & = \downarrow \\ (\downarrow \text{PCASE}) & =\text{c on} \\ /(\uparrow \text{FOR OBJ} & = \downarrow \\ (\downarrow \text{PCASE}) & =\text{c for} \\ / \ldots /\} & \end{array}$$

the complex function of a PP can be determined by leaving part of the function assignment variable:

$$\begin{array}{lccc} \text{VP} & \rightarrow & \text{NP} & \text{PP} \\ & & & (\uparrow (\downarrow \text{PCASE})) = \downarrow \end{array}$$

The value of the (atomic-valued) function (↑ PCASE) in the lexical entry of the preposition is then inserted for the term (↓ PCASE) in this annotated rule schema yielding an f-structure of the form:

$$\left[ON = \left[\begin{array}{l} OBJ = \ldots \\ PCASE = on \end{array} \right] \right]$$

This mechanism could also be applied in a similar way to classify adjuncts according to their thematic roles, such as LOC(ational), TEMP(oral) etc., as a function of the range of roles a preposition can encode:[3]

$$X \rightarrow \underset{\downarrow \in (\uparrow \text{ADJ } (\downarrow \text{ROLE}))}{\text{PP}}$$

The advantage of such a treatment of ADJuncts consists in the possibility to constrain the occurence of certain ADJuncts or combinations of ADJuncts in a frame-like fashion, while maintaining the set valued property if necessary.[4]

The lexical entries in LFG at the moment are mainly specified as full forms. They consist of the surface form of the terminal symbol, the category of the lexical item, which can be identical to a higher projection of a verb, i.e. a left hand side symbol of a grammar rule, and a set of annotations.

belaedt: V,	(↑ PRED)	= 'beladen < (↑ SUBJ), (↑ OBJ), (↑ MIT OBJ) >'
	(↑ SUBJ NUM)	= sg
	(↑ SUBJ CASE)	= nom
	(↑ OBJ CASE)	= nom

Among the annotations to major lexical categories such as nouns, verbs, prepositions and adjectives there ususally is a specific attribute PRED, whose value consists of a predicate name and a list of subcategorized functions.

[3]For a more elaborate discussion of the treatment of Prepositional Phrases in LFG see Netter and Rohrer 88.

[4]Some verbs of transport for example allow an ADJunct expressing a SOURCE only if there is also a ADJunct expressing a GOAL, while the reverse of this relation does not hold:

(1) Der LKW transportiert Wein nach Paris.

(2) Der LKW transportiert Wein von München nach Paris.

(3) *Der LKW transportiert Wein von München.

This relation can be easily captured by an implication of the form:

(4) { / ~ (↑ ADJ SOURCE)
/ ! (↑ ADJ GOAL) /}

It should be clear, however, that approaches along this line make sense (in terms of efficiency) only as long as the constructions of the semantic representation (as a potential filter) is done in a consecutive step and not simultaneously.

Grammatical functions occuring within the angled brackets are assumed to be semantically interpreted. Functions outside the angled brackets are syntactically subcategorized for, but not mapped onto an argument position of the predicate:

```
scheint: V,  (↑ PRED)        = 'scheinen < (↑ XCOMP) > (↑ SUBJ)'
             (↑ SUBJ NUM)    = sg
             (↑ SUBJ CASE)   = nom
             (↑ SUBJ)        = (↑ XCOMP SUBJ)
```

Variations in the grammatical relations, such as passives are captured by changing the function argument assignments. The implicit argument of passives is represented by a NULL function, which is assumed to be existentially quantified:

```
beladen: V,  (↑ PRED)        = 'beladen < (↑ NULL), (↑ SUBJ), (↑ MIT OBJ) >'
             (↑ NULL PRED)   = exist
```

The token set of grammatical functions occuring in the subcategorization list of the lexical items of a grammar defines the set of *governable functions* underlying the *coherence* and *completeness conditions* of LFG. These conditions make ensure that all and only the governable functions subcategorized by a lexical item are present in the respective f-structure.

During the ACORD project another form of lexical entries has been added to allow the description of numbers, dates and numbered names within an LFG lexicon. An example of the new notation is the following entry from the German ACORD grammar:

```
'Fahrer$s?$[0-9]+': NP,  (↑ PRED)       = "'driver$#3'",
                         (↑ GENDER)     = mas,
                         {/ #2          = s
                          (↑ CASE)      = gen
                         / #2           = "
                          (↑ CASE)      /= gen
                         /}.
```

The upper-case 'F' is needed for the correct writing during generation, but parsing is *not* case sensitive.

This entry should be read the following way. If a word starts with 'Fahrer' followed optionally by an 's' and ends with a sequence of digits, this word should be treated as an NP. Its annotated f-structure gets a generic PRED whose name starts with 'driver' and which ends with the same sequence of digits as given in the lexeme. The gender of the NP is masculine, the case genitive iff there was the 's' in the lexeme.

In general a concatenation of regular expressions is enclosed in backquote signs (‘). $ separates the single regular expressions, #< N > refers in the context of an f-descript-

ion to the Nth segment of the lexeme.

The basic regular expression is one character. The regular expressions itself may contain the following special characters:

? the expression before is optional

(start a subexpression

) terminate a subexpression

[start a set of characters

] end a set of characters

– inside a set between two characters a shorthand of the range of characters given by the left and right character

* repeat the expression before 0 or more times

+ repeat the expression before 1 or more times

After the formulation of the original version of LFG an alternative mechanism for the treatment of non-local dependencies was introduced, called *functional uncertainty.*

The technical foundations for this alternative consisted in an extension of the well-formedness definition of functional rule schemata. Whereas in the traditional LFG rule schemata had to consist of a finite string of attributes, functional uncertainty presupposes that rule schemata may also be constructed with expressions representing a regular language over attributes[5].

$$\begin{array}{lcll} \text{S1} & \rightarrow & \text{NP} & \text{S} \\ & & (\uparrow \text{XCOMP}* \{\text{OBJ/OBJ2}\}) = \downarrow & \uparrow = \downarrow \end{array}$$

This means that the specific function of a dislocated constituent does not have to be determined indirectly by means of relating it to another position in the c-structure, but can be established on the basis of f-structure paths denoted by the (potentially non-finite) set of strings specified by the regular expression. Which path of attributes out of this set, i.e. which grammatical function at what depth of embedding in the f-structure, is the correct one may be uncertain to a given moment, but can be easily decided on the basis of general well-formedness conditions over f-structures. As a consequence, the introduction of empty nodes in the c-structure becomes altogether obsolete. [6]

[5] In the given implementation functional uncertainty is not fully included, but simulated in an obviously limited way by a finite disjunction of rule annotations specifying possible f-structure paths.

[6] For more elaborate discussions see Netter 86, Netter 87 and Netter 88.

5.2.4 Common Modules of the Stuttgart NL part

There are three modules, which are commonly used by the modules mentioned so far :

- a general pretty printer for PROLOG, needed for dumps
- a module for managing finite automatons
- a reader for LFG grammar files

A general pretty printer for PROLOG During the development of the modules, it became necessary for test purposes to have a pretty printer for general acyclic PROLOG structures. Starting from a relative simple version the pretty printer now is also used for the listing of PROLOG clauses and databases. This feature is used for making the PROLOG dumps of the grammar for the parser and the generator. The reason to use the pretty printer instead of the build-in *listing* predicate is the PROLOG dependency of the resulting output from *listing*. For some special cases it would not be possible to read a dump file made with one PROLOG dialect into an other, since their scanners behave slightly different.

The pretty printer as completely given in the file

$ACORD_STUT/generation/pppred.pl

never changes its input structure, although the output may be a different structure.

Obviously, the pretty printer is useless for other programming languages than PROLOG.

There are several top-level predicates for the different kinds of pretty printing, where all of them are special cases of exactly one predicate call :

```
ppp(Form,Type,Params)
```

The following predicates are defined

predicate	corresponds
ppp(Form,Params)	ppp(Form,std,Params)
ppp(Form)	ppp(Form,std,[])
pp(Form,Params)	ppp(Form,[end_dot=false | Params])
pp(Form)	pp(Form,[])
pp_fstruct(Form,Params)	ppp(Form,[fspp=on,closed_list=on | Params])
pp_fstruct(Form)	pp_fstruct(Form,[])
pp_list(Form,Params)	ppp(Form,[closed_list=on | Params])
pp_list(Form)	pp_list(Form,[])
pp_listing(Form,Params)	ppp(Form,clause,Params)
pp_listing(Form)	pp_listing(Form,[])
pp_listing	pp_listing(_)
pp_list_db(Form,Params)	ppp(Form,db,Params)
pp_list_db(Form)	pp_list_db(Form,[])

The standard form of ppp without any parameters takes some PROLOG structure as input and prints it in a PROLOG readable form to the standard output. pp behaves in the same way, but does not print a dot at the end. `pp_list` is used to guarantee that there are never restvariables in PROLOG lists. `pp_fstruct` prints the internal *f*-structure format of the phrase generator in a more compact and human readable form.

`pp_listing` takes either an atom, atom / integer, a general functor or a list of such entities, and performs a pretty printing of the named clauses. In case of a variable as input, all currently defined clauses are listed. Using a general functor only those clauses which have a solution for a call are selected.

`pp_list_db` takes the same type of input as `pp_listing`, but it lists always facts. Since this means that original clauses are represented as a list of facts, this predicate may run into an infinite loop, if a clause has infinite many solutions[7].

Except for the top-level predicates all other predicates are treated as local to the file and start with the prefix

```
imsv_pp
```

Customizing the pretty printer The pretty printer is written in a way which easily allows to transform structures with respect to their normal PROLOG representation. The transformation itself is described using

```
imsv_pp_filter(Name_of_structure,
               Arguments,
               Output_Structure,
               Variable_List)
```

[7] Also do not run this predicate on the pretty printer itself.

If this clause succeeds, the Output_Structure replaces the current structure for pretty printing, where the pretty printer operates recursively on the new structure. Doing so, it is even possible to suppress parts of an input structure.

The activation of a transformation can be triggered using :

```
imsv_pp_db_param(Flag,Value)
```

The flags can be set explicit in the list of parameters given to ppp. For each known flag exists a default, which may also depend on a flag setting independent of the pretty printer. For the access to external flags

```
imsv_pp_ext_flag(Flag,Value)
```

is used.

The defaults itself are described by

```
imsv_pp_db_param_default(Flag,Value,Call)
```

If `Call` is of the form

```
lambda(X,C)
```

`C` is evaluated. If it succeeds, `X` becomes the default value. In all other cases `Value` becomes the default value.

The version of the pretty printer delivered with ACORD uses the following flags:

Flag	Default	Description
start_indent	0	the left margin
tableng	8	spaces corresponding to a <tab>
maxcol	200	the right margin for standard pretty printing
clausecol	79	the right margin for pretty printing of clauses
init_nls	1	new lines before pretty printing starts
exit_nls	2	new lines after the pretty printing
end_dot	true	print a dot at the end of a structure
closed_list	off	rest variables in lists are suppressed if 'on'
fspp	off	print a compact form if 'on'
term	on	print an abbreviation if 'off'
occ	on	suppress occ structures if 'off'

Each flag can be set in the Params list by an entry

```
Flag = Value
```

Using concatenating finite automatons As already mentioned the format of lexical entries used in Stuttgart allows to describe a lexeme by a concatenation of regular expressions, and to refer to the single parts in a uniform way.

These special descriptions are translated during reading the grammar to unique so called concatenable finite automatons. Unique means that if the same regular expression is used twice, there will be only one internal finite automaton. Concatenable means that a given regular expression always has to match the head of a given string. The rest of the string can be used as input to another finite automaton.

The directory

$ACORD_STUT/fina

contains all files for the management of the needed automatons. The **Readme** file in this directory explains the most relevant predicates.

To load the package into PROLOG use the file

fina_load+

The finite automatons generated from a regular expression have the general form

FA(CString,NormString,CRest,NormRest,CHead,NormHead)

with

CString Upper- and lower-case letters match exactly

NormString Upper-case letters are converted to lower-case.

CRest The rest of CString after the matching

NormRest The rest of NormString after matching

CHead The head string matched by the finite automaton in the exact form

NormHead The head string matched by the finite automaton in the normalized form

The automaton can start with every given type of string. Note that the strings are represented as PROLOG strings.

Although a finite automaton could run with all arguments given as variables, such a call will be in an infinite loop if it can match indefinitely many strings.

The finite automatons are generated using a slightly modification of the algorithm given in Aho et al. 86. The difference to this algorithm is that the basic item of a transition node is a character set instead of a single character. This modification allows a more compact representation.

Given a regular expression as a PROLOG atom in a format similar to the one used in the UNIX **egrep** command[8] one can build a finite automaton with

fina_get_finite_automaton(RegularExpression,Name)

where **Name** is the name of the six placed PROLOG predicate to call the particular automaton.

The automatons needed by the parser and the generator have to concatenate several finite automatons for matching a string and doing the appropriate segmentation of the string, respectively concatenating a string according to it's segments determined by the generator.

To build such a so called conc automaton call

fina_conc_automatons(List,Table,Call)

where

List is a sequence of atoms or variables referring to finite automatons as given in **Table**

Table describes the association between (segment) variables and regular expressions by a list of elements of the form

Var : Name_of_finite_automaton

Call is the generated PROLOG call for the conc automaton

Call is a four placed predicate of the form

CA(NormString,CString,NormSegments,CSegments)

where

[8]The restriction is that every characterset which can be expressed must be explicitly given. This means, there is no meta variable for 'match any character' and no complement set.

NormString is a PROLOG string with no upper-case letters

CString is a PROLOG string which represents the correct writing

NormSegments are the segments of NormString matched by the automaton

CSegments are the segments of CString matched by the automaton

Segments are represented as PROLOG lists of PROLOG strings.

The automatons can be saved with

fina_dump(File).

For other manipulations of the automatons see the mentioned **Readme** file.

A reader for LFG grammars The LFG-grammar-reader and -parser, which has been implemented as part of the LFG-parser compiler, is also used in other modules of the system.

It can be loaded by consulting the files[9]

$ACORD_STUT/tomita/compile/propatch	a patch file
$ACORD_STUT/tomita/compile/readin	the reader for LFG-entries
$ACORD_STUT/tomita/compile/parse	the parser for LFG-entries

These files define the predicates **read_entry/1** and **parse_entry/2**. **read_entry/1** reads an LFG-entry from the current input stream into a list-representation, whereas **parse_entry/2** converts such a list into a PROLOG-term that represents the logical structure of the entry.

read_entry/1 reads from the current input until it finds either a "." which is not quoted and not part of a comment, or end_of_file. The characters read are grouped to tokens, which are collected into a list according to the following rules:

- Alphanumeric characters and the underscore (_) are interpreted as part of a potentially multi-character word, also the quote-sign (') if it follows directly one of the other characters. Words consisting of such characters are represented as PROLOG-atoms in the output.
- A sequence of arbitrary characters starting with and ended by a single quote-sign (') is represented as a PROLOG term of the form **quote(Atom)** in the output, where **Atom** consists of the characters between the quotes. If the sequence contains double quote-signs, these are interpreted as a single quote.

[9]Since they were not intended to be loaded into the complete system, the predicates in these files do not have a typical prefix to avoid accidental name clashes.

- A sequence of characters embedded between backquote-signs (‘) (a regular expression) is represented in the output as a PROLOG term of the form **backquote(List)**, where List is a PROLOG-list of ascii-string-representations of the components of the regular expression, which are separated by dollar-signs.
- A semicolon starts a comment, which is ended by a newline-character. A comment is not represented in the output of **read_entry**.
- Any sequence of nonprinting character is interpreted as token separator, it is not represented explicitly in the output.
- Any other printing characters are interpreted as one-character tokens, which are represented as PROLOG-atoms in the output.

The predicate **parse_entry/2** interprets the output of **read_entry** and produces a PROLOG-term according to the following type description[10]:

```
Entry --> gram_rule(Nterm,ConstituentList)
        | lex_entry(TermOrRegTerm,node(Nterm,_),Annotations,RegTab).

ConstituentList --> []
        | [Constituent|ConstituentList].

Constituent --> const(Nterm,Annotations)
        | fac(ConstituentList)
        | rep(ConstituentList).

Annotations --> []
        | [Annotation|Annotations].

Annotation --> eq(Arg,Arg)
        | neq(Arg,Arg)
        | eqc(Arg,Arg)
        | ele(Arg,Arg)
        | '~'(Arg)
        | '!'(Arg)
        | or(Annotations).
```

5.2.5 The Parser Compiler

The purpose of the parser compiler is to read grammar- and lexicon-files written in LFG-syntax and to transform the entries into a form suitable for the LFG-parser. Within the ACORD-software, the parser compiler is never loaded into the complete system, instead,

[10] Uppercase words stand for non-terminals.

the result of the compilation is written onto a dump-file, which again is loaded by other components of the system.

The parser compiler is loaded by consulting the file

$ACORD_STUT/tomita/compile/start.acord

It is activated by calling the predicate
`compi(ListOfLFGfiles,DumpFile1)`
where the first argument is the List of all LFG-files to be compiled, and the second is the name of an intermediate dump-file.

The result of the compilation is written to a dump-file by calling
`write_grammar(OutputFile)`

The output of the parser components consists in the definitions of the following predicates:

file_compiled/1: An enumeration of the source files that have been compiled.

grammar_symbol/1: An enumeration of the nonterminal symbols appearing in the grammar, including those that have been generated by the parser compiler when grammar rules containing optional or kleene-star constituents were transformed into a set of ordinary grammar rules.

gov_fun/1: An enumeration of grammatical functions that appear in the semantic form of some pred-feature. Those functions are treated in a special way by the parser, since they are subject to tests for syntactic coherence.

start_symbol/1: An enumeration of nonterminal symbols (subset of **grammar_symbol/1**) that are acceptable as topmost node in a valid derivation.

status_flag/2: An enumeration of flags that indicate the status of the system. (Not used within the acord software).

'lex$'/5: The internal representation of the compiled lexical entries. The five arguments are:

1. The word
2. The syntactic category
3. A list of PROLOG-variables that have to instantiated by unique values when the entry is used.
4. The *f*-structure associated to the word
5. The source file where the entry is defined

gram_rule/6: The internal representation of the compiled grammar rules. The six arguments are:

1. The name of the rule (which is generated by the system and referred to in the parser tables)
2. The syntactic category of the left-hand-side of the grammar rule
3. The syntactic categories on the right-hand-side of the grammar rule
4. A list of PROLOG-variables that have to instantiated when the rule is used. The first variable refers to the top-level-label of the left-hand-side constituent, the other variables to those of the constituents on the right-hand-side of the rule.
5. The *f*-structure associated with the rule
6. The source file where the rule is defined

table_accept/1: An enumeration of those states in which the parser can accept the input.

table_goto/2: For each state an enumeration of entries in the goto-table. Each entry is a pair `category:state`.

table_reduce/2: For each state an enumeration of grammar rules that can be used for reductions in this state.

follows/2: For each syntactic category a list of those categories that could follow in a valid derivation.

5.2.6 The LFG-parser

The purpose of the LFG -parser is to analyse sentences according to the given LFG-grammar and find the *f*-structures that are associated to them. Within the ACORD-system, it is assumed that the grammar has been compiled by the LFG-parser compiler and that its internal representation is stored in a dump-file. The parser can then be loaded by consulting both the file

$ACORD_STUT/tomita/parse/start.acord

and the dump file produced by the parser compiler.

The parser is activated by calling the predicate

```
stutom_parse(Input,Cat,Result).
```

where the first argument is the input-sentence given as a PROLOG-list of words (terminated by "."), **Cat** is the syntactic category of the input (in the german grammar, **s1** is

used as root-category for sentences), and X is the output f-structure in a representation suitable for futher processing in the other components.

If the sentence can be analyzed, the output consists of a list of attribute-value-pairs which are built with the functor '='/2. The values are either atomic values or lists of attribute-value-pairs. Each list contains as its last element a term of the form stut_index(N), where N is a number which is unique for the respective f-structure. This number makes it possible to distinguish the case where f-structures are identical by virtue of an equation (token identity) from the case where the structures accidentally contain the same information (type identity). As an example, the following interaction could take place if the german grammar is loaded into the parser:

```
?- stutom_parse([der,lkw,faehrt,'.'],s1,X).
X = [pred = fahren([subj]),
     pos = 3,
     subj = [pred = truck,
             gender = mas,
             num = sg,
             pos = 2,
             spec = def,
             wh = minus,
             stut_index(1)],
     topic = [pred = truck,
             gender = mas,
             num = sg,
             pos = 2,
             spec = def,
             wh = minus,
             stut_index(1)],
     wh = minus,
     stut_index(2)]
yes
?-
```

The analysis is performed in three steps:

1) **Context free analysis:** Using the context free skeleton of the LFG-grammar, a parse forest representing all possible context free analyses of the input is built by a table-driven shift-reduce-parser (see Eisele and Schimpf 87 for details). The forest is stored in the database under the dynamic predicate packed_node/5. The entries have the form

```
packed_node(Cat,From,To,RHSs,Num)
```

where Cat is a syntactic category that has been recognized, From and To specify the

range in the input string that is covered by this category, RHSs is a list of possible rule applications that led to `Cat`, and `Num` is the number of references to this node from nodes higher in the tree. The rule applications are terms of the form `n(Prod,RHS)`, where `Prod` is the name of the grammar rule that has been used and `RHS` is a list of terms of the form `packed_node(Cat,From,To)`, which refer to subnodes. Subnodes which refer to empty constituents are represented as - in such lists.

2) Evaluation of *f*-structures: In a second step, the parse forest constructed in the first step is traversed in a top-down manner and for each node, a set of *f*-structures is computed. For toplevel nodes of the derivation, this *f*-structure is stored in the database under the predicate `found_fs/6`. Here, the arguments are

1. A unique number
2. The syntactic category
3. The *f*-structure, where all consistent branches of disjunctions are represented, even if they contain unsatisfied existential constraints
4. The *f*-structure, where all incomplete branches are filtered out
5. The c-structure of the derivation as a prolog term, where also artificial nodes generated by the parser compiler are represented
6. The same c-structure, where all artificial nodes are eliminated

The *f*-structures computed in this step may contain disjunctive values as described in Eisele and Dörre 88 and Eisele 87.

3) Adaption for further processing: In a third step, the resulting *f*-structures are transformed into a representation that is suitable for further processing in the ACORD-system (see the example above). In the ACORD-system, not all features contained in the LFG-parser are exploited. For instance, disjunction embedded in the *f*-structures is transformed into top-level disjunction via backtracking, since some of the other components do not yet support the representation of embedded disjunction.

5.2.7 The Semantic Constructor

Integration with parser The semantic system gets integrated into the parsing system by consulting the LOAD-file

$ACORD_STUT/meaning/tom_update

which reconsults all required files.

In the implementation the non-built-in Prolog predicates used are prefixed by *stut_* to avoid accidental name clashes with predicates used in other components of the complete system (eg parser). In order to keep the system self contained for reasons of modularity it was avoided to make use of predicates defined in other components though sometimes they would be available there. But this is an accidental fact and cannot be relied on.

A detailed documentation on the files and the defined predicates is available in the file

$ACORD_STUT/meaning/doku.tex

f-structures The program expects the f-structure graph expanded to a tree structure in the form of a PROLOG list of attribute-value-equations *Attribute* = *Value*, where *Value* can also be a list of attribute-value-equations. Reentrancy is indicated in the expanded structure by a unique (numerical) index associated with each non-terminal node of the graph, that is, nodes with the same index are regarded as identical. This index has the form *stut_index(Number)*. Also, for those nodes having a PRED-edge there has to be an associated feature *pos* with numerical value that indicates the position of the associated lexeme in the surface structure.

Call The top-level-predicate for calling the semantic constructor after parsing is

`stut_semantik(F_Structure,InL).`

For running the semantic constructor alone with the parser, the predicate `stut_pars` can be used, which reads a sentence, parses it and prints the InL. Various flags can be set to influence what is printed and how it is printed which are described in the file `tom_flags`. For using the parsing system including the constructor in other systems (eg. ACORD), there is the predicate

`stut_pars1(s1,Input,InL)` where

s1 the start category of the grammar for sentences

Input the sentence in the form of a list of words ending with a fullstop.

InL the resulting InLfor the sentence

The algorithm The algorithm basically runs through the f-structure recursively in a top down fashion computing for each node a list of partial semantic structures which gets reduced to a single semantic structure by functional application. Terminal nodes get their meaning from a semantic lexicon. The partial meaning structures internally used are quintuples of the form

`[Index,Occ,Abstrl,InL,Over]` where

Index an InL term being the index of the structure

Occ the occurrence information for the structure

Abstrl the *abstraction list*, consisting of triples of the forms

- `[Index,InL,Occ]`, meaning that the argument has to be a partial InL with index `Index` and occurence information `Occ`.
- `[Index,GramFunction,Occ]`, where `GramFunction` designates the grammatical function whose semantics supplies its `Index` as argument for InL-predicates

InL an InL expression which might contain variables for expressions bound in the abstraction list

Over a relict from an older implementation where it was used to restrict functional application. It is now superfluous.

The algorithm standardly starts out from the PRED of an f-structure, and computes the set of meanings of its subcategorizing functions and of other features present. But in some cases it can be necessary to deviate from the standard order of processing, mostly due to some special requirements, ACORD puts on the semantic representations, where we choosed to complicate the mapping to semantics instead of complicating the grammar. This requires to put in additional rules for special kinds of f-structures. Changing the grammar could therefore require to adapt these rules too. At present, there are special rules for the following cases:

- Sentence Mood
- Subordinate Clauses
- Relative Clauses
- Possessives
- Coordinations
- Adjuncts

There are also some additional limitations on the constructor with respect to what kinds of f-structures can be processed in its current version.

- Due to requirements of the interface structure for the RESOLVER the f-structures are required to contain a subject-edge, and so the constructor can directly be used only for sentential f-structures though in principle it could construct (partial) semantic representations for f-structures of any category.
- The paths for subcategorizing functions are limited to a length of one, though LFG allows also deeper embeddings. But this is not used in the ACORD grammar, and so the restricted version is sufficient.

- The recognition of reentrancy is limited to non-terminal nodes, and fails in the case of terminal nodes. This can yield wrong results especially when single PRED-values can be reached on different paths (according to the grammar), because the f-structures currently do not represent these paths. In the ACORD grammar these cases do not occur.

It is intended to integrate most of the building of semantic structure into the parser, similarly as has been proposed in situation theoretic frameworks (Fenstad et al. 87, Halvorsen 87, Halvorsen and Kaplan 88). Then syntactic and semantic analysis could be done in parallel, and it would enhance the usability of the semantic constructor for other applications. The feasiblity of this enterprise requires better tools for grammar development and grammar description, which have not been integrated into the system up to now.

5.2.8 The f-structure generator

This is an ACORD specific module, transforming SynInLs into underspecified f-structures. Due to the history of this module there are several different prefixes for predicates which are local to the module :

- syn2fs_
- inl2fs_
- inlunify_

The only top-level predicate is

syninl_to_fstructure(SynInL,DefaultCategory,List_of_FStructures)

Where *SynInL* must be non variable, the *DefaultCategory* may be variable or of one of the values

- s1
- np
- pp

and the *List_of_FStructures* is the result of the call, which means that this argument has to be a variable when the predicate is called.

Internally a unification procedure is used, which is able to treat simple disjunctions on atomic values, and negation on atomic values and forms, e.g. a value is no f-structure,

set, etc. This program is called by inl2fs_merge(FD_In,FS_Out) where FD_In is either an *f*-structure itself or an *f*-description in form of a PROLOG structure.

For complex disjunctions, e.g. if values depend on each other, a default mechanisms with levels is used. This means that defaults and conditions are evaluated after most of the final *f*-structure is instantiated. The order of the evaluation depends on level indicators, which are represented as numbers in increasing order.

The unification procedure is based on a unification algorithm described in Kohl 88. Since the implementation of the algorithm described there has some disadvantages at least for C-Prolog[11], only the simpler parts had been implemented for ACORD[12].

An *f*-structure produced by the unification procedure has to be changed before it can be used by the phrase generator, since it contains information needed only by the *f*-structure generator and uses an other internal format than the phrase generator. This filter module may be useful as basis for the transformation of other kinds of internal *f*-structure formats, since most of the structure specific elements of the code are described by access predicates instead of the concrete substructure. This data abstraction is used through most of the code for the *f*-structure generator, and had been very useful for the development of the generator.

5.2.9 The phrase Generator

The phrase generator is a general tool for generating phrases from *f*-structures. It consists of

- a compiler, which translates LFG grammars into an internal representation for the phrase generator,
- the kernel of the phrase generator
- and some predicates
  - to manage grammar dependent settings,
  - to list information extracted from the grammar
  - to transform other *f*-structure representations to the one used internally.

The phrase generator is described in detail in Momma and Dörre 87. The compiler of the phrase generator is described in detail in Deliverable T2.10.

The phrase generator is used in Stuttgart not only for ACORD but also for the development and debugging of other LFG grammars. So the current version is tested for

[11]The algorithm uses cyclic stuctures, and the implementation turned out to be too slow.

[12]There is a modified implementation in progress, which uses acyclic structures and explicit pointers, to realize all parts of the mentioned algorithm, but there was no time left, to integrate this version into ACORD.

C-Prolog as well as for QUINTUS PROLOG. In principle the generator can be used in SICSTUS PROLOG and in ARITY PROLOG as well.

There are several restrictions for the phrase generator, which arise from the current design :

- grammars, which produce as parse result more than one root f-structure, will cause the phrase generator to fail.
- a given f-structure has to be completely specified with respect to the hierarchy of all sub-f-structures.
- it must be specified explicitly, which atomic valued features may be introduced during generation.
- it must be specified explicitly, which inequations on atomic valued features should be treated like defining equations on the complement set.
- the compiler does not introduce a trivial equation into a grammar rule automatically, if there is no reference to the f-structure associated with a node. There are LFG systems, which would behave this way.

The last point is a decision, which is made only for reasons of clarity. Since the compiler recognizes this case, and gives the user a warning, it is easy to change this behaviour, if one wants this.

The first two points have their reason in the fact that it is not allowed to introduce f-structures during generation to avoid infinite loops during the generation of one phrase.

The remaining two points have their reasons in the current implementation of the compiler and the restricted expressive power of the internal representation of f-structures. The current implementation of the introduction of atomic valued features could lead to an infinite loop, if *every* attribute would be allowed to introduce.

The work on the phrase generator will be continued in Stuttgart. Beside the improvement of the performance, which depends much on the improvement of the compiler, work will be done, to overcome the necessity to specify explicitly grammar dependent information and to determine from the grammar at compile time, which f-structures may be introduced during generation under which conditions[13].

The compiler currently only compiles a complete grammar. It collects no information which would make it possible, to change only parts of a compiled grammar. The improvement of the compiler will change this also.

The compiler is usually called by :

[13] One may use f-structures as complex atoms.

tglfg_compile(List_of_grammar_files,Dumpfile).

Each file in the List_of_grammar_files is read into PROLOG, the compiler is started, and after finishing, the dump of the compiled grammar is written to Dumpfile.

There are also forms of the call, which does not automatically write the dump[14], or make use of a PROLOG representation of the grammar. For details see the file g_comp.pl.

Except for some of the predicates which are intended to be used interactively, all predicates of the phrase generator module start with tglfg_. The exceptions are

allow(..)	allow an atomic featured value to be introduced
allow_all	allow all atomic featured values to be introduced
allowed	list the introduceable atomic featured values
disallow(..)	undo allow for an attribute
disallow_all	undo allow for all attributes
disallowed	list all atomic featured values, which cannot be introduced
inlex(Atom)	check, whether Atom is in the lexicon
save_allowed_attributes(File)	save these attributes to the file

`allow` and `disallow` are either one placed or two placed predicates. The two placed form takes exactly one attribute value pair, while the one placed form is either a single attribute, represented as an atom, an attribute value pair, using the comma operator, or a list of those entities.

All these predicates are defined in g_inter.pl, so problems with name clashes can be solved, by not loading this file.

Analysing grammars with the phrase generator For a new grammar it may be the case that it overgenerates. If underspecified f-structures are used, adjustment of the grammar specificic information is necessary. In both cases it may become necessary to trace the phrase generator. There are only few predicates whose call results are of interest, and which should be spied.

The call predicate is

tglfg_gen_bf(Category,FStructure,Sentence).

where `Category` is the start category for your grammar, `FStructure` the input f-structure, and `Sentence` the result as a list of PROLOG atomics.

The first interesting predicate call is

[14]This can be done explicitly by tglfg_dump(File).

tglfg_match

which selects for a category the associated subcategories and their annotated f-descriptions. The following call of

tglfg_match_exprs

checks, whether the annotations are valid with respect to the given input f-structure. It should be easy to determine, whether the result is expected or not.

If everything has been derived successfully you will reach

tglfg_gen_level

which checks first some constraints, before it continues with the next level. Take a careful look on the current c-structure tree, to determine, whether it is the expected one. Since repetitions of a sequence of rules are expanded only at the second try, it may be easier to skip the details of a derivation.

If there are all f-structures consumed, tglfg_gen_level checks the remaining constraints, and produces in the case of success the output list.

5.2.10 A Local Test Embedding

To be able to test some of the code developed at Stuttgart independent of the rest of the ACORD system, a small test embedding had been written. This module is defined in the file

$ACORD_STUT/demo.pl

and the complete system can be loaded into C-Prolog by

consult(demo_load).

C-Prolog must be called with the following settings :

- −h2048
- −g1024
- −a324

- –l1024
- –x32

otherwise it will run out of space somewhere. It is recommended to make a saved state after loading the system, since it takes several minutes, to load all the files. To make a saved state simply call

make newdemo

Calling

demo.

a read-eval-loop is started, which allows to give either a command or to parse a sentence. Everything, which cannot be interpreted as a command is interpreted as a sentence. The interpretation always reads until a dot or question mark is detected. Be aware that the test embedding has a lot of inconviniences and is only intended for simple testings. Although there is a minimal help facility, which allows to get information about commands and the meaning of various flags, nobody should expect to work with a good user interface.

If the input was a parseable German sentence, the semantic constructor is called. Afterwards there is an optional call of the RESOLVER[15] The resulting InL[16] is transformed to a SynInL, which is used as input for the f-structure generator. The resulting f-structures are used for the phrase generator, which is called on each single f-structure.

The user can control by various flags, whether intermediate results are displayed and if the RESOLVER should be used. It is also possible to control whether a submodule should be able to backtrack or not. For details call 'help' in the demo-loop.

5.3 Relation to Other ACORD Components

For insertion in the whole system first make sure that the dumps and necessary loadfiles are up to date, by calling

make install

when in the directory $ACORD_STUT.

[15]The use of the resolver is restricted, since it depends in general also on the KB.
[16]Either a resolved or an unresolved one, depending on the use of the RESOLVER.

If one changes the grammar in $ACORD_STUT/gram call

make dumps

in the $ACORD_STUT directory.

After that it is sufficient to load

$ACORD_STUT/sys_nl_ger

into the ACORD system using *consult*, to get the Stuttgart NL part.

For a stand-alone version use the file demo_load[17].

The two mentioned load files and the makefile in $ACORD_STUT can be used with C-Prolog only. For other PROLOG dialects all the makefiles in the subdirectories of $ACORD_STUT, demo_load, sys_nl_ger, and sys_tg_ger[18], must be slightly modified. The makefiles in the subdirectories of $ACORD_STUT already produce load files, which use absolute path names, for C-Prolog and QUINTUS PROLOG as well. See also below **5.4.**

5.4 Potential of Development

From the various parts of the software described above, only three components are specific to ACORD :

- the lexicon part of the grammar
- the semantic constructor
- the *f*-structure generator

All other parts are tools independent of the ACORD application and already used for other purposes in Stuttgart.

Being specific to ACORD means for these components only that they cannot be used without changes in other systems, but at least those parts, which are independent of specific structures, predicates and words, used in ACORD can be useful outside ACORD.

[17] See above.

[18] Which is loaded by sys_nl_ger.

Development in the system

As far as the ACORD system is concerned, the usage of tense or extensions to the lexicon would force some additions in the ACORD specific parts. The f-structure generator could be changed a bit, if a more general unification algorithm is ready for integration. This would concern the description of the entries for the mapping from SynInL to f-structures and the current treatment of complex disjunctions and defaults.

Development as an independent tool

Together with various extensions in work to the syntax for LFG grammars as used in Stuttgart, parser and phrase generator will be enabled to treat new forms of grammar descriptions efficiently. This concerns the usage of templates as well as an interface to a morphology component.

For the phrase generator most of its efficency depends on the quality of the compiler. So most improvements are expected by extracting more information from an LFG grammar by the compiler, to allow on the one hand partial recompilation of a grammar, to speed up grammar development, and on the other hand to reduce lexicon access during generation to a minimum, and to avoid grammar dependent predicates, which have to be defined explicitly by the user.

Changes to the phrase generator concerning the expressive power of its internal f-structure format are in progress. Together with the improvements of the compiler this would allow to avoid completely user defined grammar dependent predicates during generation.

Further work will be done to integrate the algorithm from Jürgen Wedekind as described in Wedekind 88 into the phrase generator. This seems to be reasonable, since this algorithm can be seen as an extension to the one the generation is currently based on.

The module which treats finite automatons can be extended to treat finite state transducers.

For parser and phrase generator work is in progress at Stuttgart to have better user interfaces, which uses both components for grammar development.

For machine translation both parser and generator must be enabled to work with more than one grammar in the same process.

Possible translations into other languages

Although this is possible in principle for every computer language, for none of the components a *direct* translation into other languages than those similar to PROLOG

is recommended, since the flow of control, unification and backtracking capacity had lead to some constructions specific for PROLOG. Unless one has a tool which translates PROLOG into another language automatically, it would be easier to reimplement the algorithms, the code is based on.

For PROLOG dialects which use the same syntax as in Clocksin and Mellish 84 it should be easy to adapt the code.

It is highly recommend to use a PROLOG which compiles the code, to use parser or generator. Our experiences with QUINTUS PROLOG had shown that the parsing and generation are about ten times faster than in C-Prolog, if the code is compiled.

Possible combinations with other components of the system

For the rest of the ACORD system there are two separateable units :

- parser + semantic constructor
- f-structure generator + phrase generator

Both units share the reader of the tomita parser, the module for finite automatons, the LFG grammar[19] and the pretty printer.

The architecture of the ACORD system allows to exchange each unit by a unit with the same functionality and the same interface calls.

[19] But both have their own dump file for the grammar with a different format.

Chapter 6

Protolexicon

Jonathan Calder,
Mike Reape
ECCS

A prototype implementation of a Lexicon Development System is presented in this chapter. It allows the succinct representation of lexical information by means of a system of defaults over the syntactic and semantic information associated with a lexical entry.

6.1 Scientific Background and Development

6.1.1 Arguments for a General Lexical Processor

Constructing lexicons is one of the most time-consuming parts of a computational linguistics project. The *Protolexicon* is a prototype implementation of a general lexical processor, by means of which a user may generate lexical entries tailored to particular linguistic theories on the basis of dictionary style lexical definitions. More details on the technical background to this work are given in Deliverable T 1.12.

6.1.2 General Design

The overall design of the current implementation is that of a preprocessor. That is, we expect the user to provide a lexicon, together with the information required by the system to interpret that lexicon. The component then uses the user's definitions to

construct a lexicon to be used by a particular grammar. The information given by the user is of the following types:

- PIMPLE templates which provide a set of *defining terms*,
- a network showing default values for defining terms and relationships between defining terms,
- lexical rules and
- morphological tables describing the inflected forms of words

In fact, most of this information is likely to be constant across lexicons in different application domains. The only kind which is likely to differ is that to do with relationships between defining terms. Likewise, the defining terms will be similar across similar grammatical theories.

The language in which the network of templates is defined is inspired by work in Systemic Grammar (e.g. Kress 76). We assume that every defining term is associated with a *dimension*, that is it represents a choice from a larger group. Thus one statement used in the current implementation is

```
fulfils(transitivity, transitive).
```

interpreted as saying that `transitive` is a defining term which specifies the behaviour of the lexical item with respect to the abstract phenomenon or *dimension* of `transitivity`. The statement

```
requires(verb, [transitivity, aspect, agreement, morphology,
                sortal_restr, stress_type]).
```

says that any entry defined as a `verb` must be further specified for information to do with transitivity (among other things). Finally, the statement

```
default(transitivity, intransitive) :- verb.
```

means that, if an item is specified as a `verb` and is not defined by some term which gives information about transitivity, the value of `intransitive` may be assumed.

The lexicon is processed to arrive at a lexicon for a particular linguistic theory, by evaluating lexical entries with respect to the user-supplied information. Evaluation proceeds by the following steps. The templates are processed using the standard PIMPLE mechanisms discussed above in 1. The network of relations between defining terms is analysed for certain properties required by the interpreter, namely that there are no cyclic definitions. The morphological tables are then analysed to determine the relationships that hold between them. The important relation in this context is that of subsumption. That is, we have to determine of each pair of tables whether one is more specific about the lexical items whose behaviour it defines. The user-supplied lexicon is then loaded and information which is predictable by the default rules of the network is added. The final stage consists of constructing the full set of lexical entries defined by the lexical items in conjunction with the morphological tables and lexical rules. The resulting lexicon is therefore a *full-form* lexicon, in that no morphological processing is done on-line.

6.2 Technical Description

The Protolexicon is closely integrated with the PIMPLE system and the reader is referred to 1 for general information regarding that system.

6.2.1 Files

Source code for the Protolexicon is to be found in `$ACORD_EPI/Protolexicon/src`. File names beginning with "q" are contain routines used by the Protolexicon. A file called `MANIFEST` in this directory gives more details of the contents of the individual files.

6.2.2 Software Description: General Design

Implementation strategy We have followed the same strategy for the representation of protolexical information as used in the PIMPLE system described in 1. That is, we store Protolexicon under keys in PROLOG's secondary database. The objects recognized by the Protolexicon are

`fulfils`	Information about templates that represent a particular dimension
`requires`	Information about the dimensions implied by a particular template
`default`	Information about the defaults associated with a particular dimension
`cclass`	Sets of characters classes for morphological tables
`table`	The morphological tables

File handling The files used by the Protolexicon are the following, where `Name` is the value of the PIMPLE variable `grammar_name`.

`Name.lr` Protolexical format lexical rules
`Name.mt` Morphological tables

Note that the routines that handle lexical entries in Protolexical format are called by means of the PIMPLE handler declaration `in_file/2`.

Lexical entries Protolexicon lexical entries have the form

```
Form :- DTerm1, DTerm2, ..., DTermn.
```

where `Form` is a PROLOG atom, `DTermi` is a PROLOG atom and possibly a template name.[1]

Also allowed are templates of the form `Name(Arg)`, a parameterized template, where in the template definition `Arg` is unified with some value in the template. Such forms are used for introducing arbitrary information, for example the semantic translations of words, into a feature structure. (Note that such terms may *not* take part in any network relation). Other forms of this kind are special to the morphological Protolexicon, namely `morph_root(String)` and `morph_stem(String)` which represent string forms which may appear in the inflected forms of a particular word. If no specification is given using this mechanism, the values are taken to be identical with the string form of the entry.

A Protolexical entry is processed by applying the defaults given by the systemic network (see next paragraph) and stored as a PIMPLE object under a key formed by the concatenation of the symbol `#` and the string from of the entry and using the functor

```
basic_entry(Word, Ts, Struct, Morph)
```

where `Word` is the string form of the entry, `Ts` is the expanded set of defining templates, `Struct` is the feature structure that results from combining the information from these templates and `Morph` is a structure containing any information from `morph_stem` and `morph_root` specifications of the entry.

The network file The network file contains statements about the defining terms used by a particular grammar, as discussed above. The full range of forms is:

[1] *Possibly* because we allow defining terms not to have a definition as a template. This is useful in the situation where a specification for a particular dimension implies the presence of other features, but does not need to introduce any information itself. This behaviour depends on the value of the PIMPLE variable `ignore_unknown_templates`.

```
requires(Atom, ListofAtoms).
fulfils(Dimension, DTerm).
default(Dimension, DTerm).
default(Dimension, DTerm) :- DTerm1.
```

Apart from `ListofAtoms`, all instantiations must be atomic. In the case of the second form of the default statement, `DTerm` must be dependent on `DTerm1`, in the following sense. A defining term is dependent on another if they are in the transitive closure of the following relation:

```
dependent(DTerm1, DTerm2) :-
        requires(DTerm2, Requirements),
        member(Dim, Requirements),
        fulfils(Dim, DTerm1),
```

The predicate `q_expanded_entry(Entry, Graph)` relates a lexical `Entry` in the form described in the previous paragraph to an expanded form `Graph`, in which all possible defaults have been applied. In doing this, the first step is to restrict the lexical definition to defining terms mentioned by the network. A defining term from the definition is chosen such that it is dependent on no other term in the definition. Next a † is constructed which represents the dependent relation explicitly, and all defining terms are associated with the node in this graph. The defaults are then applied starting from the chosen node and added to the graph.

Like the description above, the code for this aspect of the system is highly procedural, and it is likely to be the case in grammars where there are interacting defaults that the system may fail to have a declarative semantics. In the current grammar this is not the case.

The protolexicon's morphological component The morphological component of the Protolexicon is comprehensively illustrated in Deliverable T 1.12. We will here concentrate on aspects of the implementation that are not covered in that document. The most distinctive facet of the morphological component is its use of partial descriptions of strings to model morphological operations. We will first give details of our implementation of strings and string unification.

6.2.3 String Unification

Many of the relations defined in the Protolexicon are defined in terms of *string unification* with identity and constraints on variables. String unification is unification with an associative function. I.e., if the operator $\cdot$ is associative then the variable substitution $\{x \leftarrow he, y \leftarrow lo\}$ will "solve" the following equation.

$$(he \cdot l) \cdot y = x \cdot (l \cdot lo)$$

The string unification predicate **su(X,Y)** is defined in the file **qsu.pl**. It is based on Siekmann's algorithm for unification for an associative function without identity (see Siekmann 75) for a description of the original algorithm). String unification is a *type-zero* unification problem. This means that there is no set of most general unifiers (*mgus*) in the general case and therefore no complete, correct and minimal string unification algorithm. A *complete* algorithm is one which finds all mgus. A correct one finds only unifiers. A minimal one finds only most general unifiers. Our strategy here is to write a fairly efficient string unification predicate which is not minimal and then use the interdefinability of subsumption and unification to filter out non-mgus from the set of unifiers. We also avoid nontermination problems with unifiers which create cyclic terms by using the occurs-check at crucial points.

The associative operator that we use is the PROLOG "." operator. That is, we require associative unification over PROLOG lists. Variables in lists are treated as *string variables*, i.e., variables which can be unified with PROLOG lists. We have also implemented a weak set of constraints on what string variables can unify with. The constraints allows disjunctive and negative "value" constraints on variables. All variables are encoded as terms of the form `X:C` where `X` is the original "conceptual" variable and `C` is the list of constraints on `X` in internal form. Constraints are encoded in open-ended lists where the "current" set of constraints is the second-last element of the list (the one before the variable last element). Therefore, an Eisele and Dörre-style constraint merging predicate can be used to merge constraint lists. Unfortunately, whenever we merge constraint stacks we must search the list for the top element.

We will now briefly describe the major predicates in **qsu.pl**.

su(X,Y) unifies two variables, a variable and a *string* or two strings. A string is a term of the form `X:C` where `X` is either a PROLOG variable or a list representing a string. `C` is the set of constraints on `X`. The primary clause is the one that unifies two strings and it calls **equate(X,Y)**.

equate(X,Y) unifies two strings `X` and `Y`. This is the predicate that is based on Siekmann's algorithm. We will describe it by clauses.
The standard base clause is:

```
equate([],[]).
```

The two following clauses ignore "." `0` is used to represent the empty list (the identity element for ").

```
equate(X,[Y0|T]) :-
```

```
  xnonvar(Y0), Y0 = 0:_, !,
  equate(X,T).
equate([X0|T],Y) :-
  xnonvar(X0), X0 = 0:_, !,
  equate(T,Y).


equate(X,[Y0|T]) :-
  xnonvar(Y0), Y0 = [_|_]:_, Y0 = YL:_, !,
  append(YL,T,Y),
  equate(X,Y).
equate([X0|T],Y) :-
  xnonvar(X0), X0 = [_|_]:_, X0 = XL:_, !,
  append(XL,T,X),
  equate(X,Y).
```

The two previous clauses basically rewrite any left-recursive terms as right-recursive terms and then recursively invoke `equate`.

```
equate([S1|S],[T1|T]) :-
  equate1([S1|S],[T1|T]).
```

Unlike the previous clauses, the `equate` clause is nondeterministic. It calls `equate1` to unify `S1` and `T1` with the occurs check and then merge their constraints. It then recursively invokes `equate` on `S` and `T`.

```
equate([S1|S],[T1|T]) :-
  \+ dvar(S1), S1 \== T1,
  equate2([S1|S],[T1|T]).
equate([S1|S],[T1|T]) :-
  \+ dvar(T1), S1 \== T1,
  equate3([S1|S],[T1|T]).
```

These two clauses "hypothesize" that either variable `S1` or `T1` should unify with a list. This is allowed if they do not have disjunctive constraints attached to them as the presence of a disjunctive constraint would mean that they could not match a list. `S1` and `T1` are required to not be equal as this is just a instance of the previous clause. `equate2(X,Y)` and `equate3(X,Y)` accomplish the "hypothesis" and unification.

```
equate([S1|S],T) :-
  xvar(S1), S1 = 0:C,
  q_ck_con(C),
  equate(S,T).
```

```
equate(S,[T1|T]) :-
  xvar(T1), T1 = 0:C,
  q_ck_con(C),
  equate(S,T).
```

These two clauses hypothesize that the variables S1 or T1 should match the empty string, 0, which is implicit everywhere. The constraints are checked to make sure that the variable can in fact match 0. Then the rest of the lists are unified.

equate1(X,Y) has already been described above.

equate2(X,Y) and equate3(X,Y) are modelled on Siekmann's algorithm. These two predicates have been partly described above. equate2 is shown here. equate3 is similar.

```
equate2([S1|S],[T1X:T1C|T]) :-
  S1 = [S11,S12]:_,
  S11 = S11X:S11C,
  S11X ~= T1X,
  q_merge_constraints(S11C,T1C),
  equate([S12|S],T),
  \+ ( xnonvar(S12), S12 = 0:_ ).
```

equate2 hypothesizes that S1 should unify with a two-element list [S11,S12]. The first element of the list S11 will be a variable which is unified (with the occurs check) with the first element of the other list T1X. Their constraints are merged and then the list [S12|S] is unified recursively with T. Rather inefficiently, equate2 then checks after the recursive unification that S12 has not been instantiated to 0. If this is the case then it wasn't actually necessary to use this clause. (This could be done more efficiently by attaching a negative constraint prohibiting the second element from being instantiated to 0.)

There are several predicates which test the type of the list element to be unified. The conditions under which they succeed are described briefly here.

cvar(X)	X is a constrained variable
cnonvar(X)	X is an atom (i.e., a constrained nonvar)
xnonvar(X)	X is ground (constrained or unconstrained)
xvar(X)	X is a variable (constrained or unconstrained)
uvar(X)	X is an unconstrained variable
unonvar(X)	X is unconstrained and ground
unconstrained(X)	X is unconstrained
constrained(X)	X is constrained
disjunctive(X)	X has a disjunctive constraint only
dvar(X)	X is a variable with a disjunctive constraint
negative(X)	X has a negative constraint only
nvar(X)	X is a variable with a negative constraint

`find_top(List,Top,Next)` returns the top constraint list `Top` of the constraint list `List` and the uninstantiated variable `Next` at the top of the list.

`q_ck_con(List)` checks that the constraint list at the top of the constraint list stack is satisfiable and adds another element to the top of the stack if one of the constraints has been satisfied (i.e., the variable to which it is attached has become ground).

`q_merge_constraints(X,Y)` merges two constraint lists together such that any constraints which are satisfied on unification are undone on backtracking. Disjunctive and negative constraints are represented in the same form as in the rest of PIMPLE.

The user may then represent partial descriptions of strings by means of the syntax:

`X`	A PROLOG variable, interpreted as a string variable
`X = C`	If C is a PROLOG atom, X is unified with that atom taken as a string. If C is of the form `'Class`, it is constrained to fall within that character class (see below). If C is of the form `{LIST}`, where `LIST` is a comma-separated list of PROLOG atoms, it is contrained to be some element of that list.
`0`	The symbol zero is taken to represent the empty string.

Character classes and morphological tables Character classes allow a simple kind of template mechanism, for use with string unification. Their use is essentially the same as that of character classes in finite-state morphology.

Their syntax is

```
class(Atom, Set).
```

where `Atom` is the name of the character class and `Set` is a comma-separated list of single characters delimited by { and }.

The syntax of morphological tables is:

```
table( Name,
       InputConditions,
       ListOfLexicalRules,
       ListOfStringForms).
```

where `Name` is an arbitrary unique identifier, `InputConditions` is a list of template names, `morph_stem` and `morph_root` specifications, and identities between partial descriptions of string. The latter are given in the syntax described at the end of the preceding discussion on string unification. `ListOfLexicalRules` and `ListOfStringForms`

are lists of equal length, the first a list of atoms naming lexical rules, the second of partial descriptions of the strings that result from application of the corresponding lexical rules.

Once the file of morphological tables has been read, they are compiled into their internal representation and analysed to determine subsumption relations using the routine `comp_tbl_sub` and subsidiary routines. Subsumption is calculated only with respect to the information contained in a table's `InputConditions`. This information is then recorded in the database for use in the processing of lexical items.

Lexical rules The syntax of lexical rules used by the Protolexicon is slightly different from that used in the standard PIMPLE system, (see **1.2.2**) namely

```
lexical_rule(Name, In -> Out, FeatureStructureIn, FeatureStructureOut).
```

where `In` and `Out` are comma-separated lists of template names and represent the mapping performed by a lexical rule in terms of defining terms.

The construction of the expanded lexicon The final stage in the process of building a lexicon from the information supplied by the user is that of computing the closure of the original lexicon with respect to the morphological tables and lexical rules. As discussed in Deliverable T1.12, a particular morphological table may only be used for the derivation of new forms if it is less specific than another table which might also be used. The term "less specific" in this context is equivalent to the subsumption relation over tables described above.

The closure of the lexicon is computed by enumerating the basic lexical entries described above and passing them to the routine `expand_entry0/11`. The first clause of this routine identifies its input and output arguments as a base case. The second clause uses a subsidiary routine which, by searching the subsumption lattice and comparing the elements with the lexical entry in hand, returns a set (typically a singleton) of tables which may be used to compute new lexical entries. Non-deterministically, choosing a table from that set, the routine `apply_table/10` enumerates the results of applying a lexical rule from the table, again chosen non-deterministically. The routine `apply_lexical_rule/5` performs the mapping specified in the Protolexicon's lexical rules.

A recursive call is made to `expand_entry0/11` using the newly computed lexical item as its input arguments. Successful rule applications are reported to the standard PIMPLE mechanism for storing definitions in the PROLOG database.

As the above procedures are embedded in a failure-driven loop, the entire search space associated with a particular lexical entry and the morphological tables is traversed.

One aspect of the implementation which requires improvement has to do with the application of lexical rules. In the current implementation, the mapping over feature terms described by a lexical rule creates a new set of templates associated with the newly created lexical item. However, given the existence of complex, structured information particularly to do with the semantics of lexical items, it was found necessary to allow a mapping defined in terms of feature structures as well. In practice, it is very easy for these specifications to diverge, in the sense that the information contained in the feature structures is incompatible with the conjunction of the templates. To get around this problem, the current implementation does not compute the feature structures defined by the templates and unify these with the feature structures associated with derived lexical items (which it arguably should do for correctness). Instead the template information is merely used to guide the computation of closure. A practical effect of this is that the current system is unable to implement the attractive analysis of passive participles which derives them from the corresponding past participle.

User-called predicates The user-called predicates are described in **1.2.2**. By default, the closure of the lexicon is not computed when a grammar is compiled. This is because lexicons of the size required for the ACORD project take a considerable while to compile. The predicate `q_load_and_close_lexicon(FileName)` is provided for the purposes of debugging. See also the next section.

Variables The following PIMPLE variables control aspects of the Protolexicon's behaviour.

`expanded_lexicon_contains_basic_lexicon`	In certain cases, the user may not wish the lexical entries he or she supplies to be included in the result lexicon. If this variable is set to `off`, only derived forms will be allowed.
`pile_all_entries`	If set to on, the process of computing the closure of the lexicon will commence as soon as all files are loaded.
`ignore_unknown_templates`	If on, it is not an error for a defining term not to have a definition as a template.
`check_output_against_system_speller`	If set to on, all computed lexical entries are passed through the system spelling checker.

Illustrative examples

Illustrative examples of the system used for the English grammar are given in Deliverable T 1.12.

6.3 Relations to Other ACORD Components

As the Protolexicon is embedded within the PIMPLE system, it has no explicit interfaces to other ACORD components.

6.4 Potential for Exploitation

The Protolexicon was developed primarily as an investigation of the possibilities of designing high-level lexicons and tools for manipulating them. As such, any further development is likely to require substantial redesign and reimplementation in the light of these experiences and developments elsewhere.

Chapter 7

Resolver

Walter Kasper
IMS

The central anaphora resolution component of the ACORD system, which is part of the DM, is described in this chapter.

7.1 Scientific Background and Development

Anaphora resolution is a notoriously difficult field, because for different kinds of pronouns and referring expressions there are different sets of until now only partly understood constraints on the relation between anaphora and possible antecedents. Also, all kinds of linguistic and non linguistic knowledge enter the process of resolution and have to interact: morpho-syntactic information, semantic information as scope, pragmatic information as discourse structure and discourse history, and conceptual and factual knowledge.

In the first prototype implementation each parser contained its own anaphora resolution component (Deliverable T1.7'(a)). This required that each parser had to manage some form of dialogue history. But these separate modules had only restricted coverage, since they could rely only on linguistic information, and had no access to non-linguistic information.

Since the languages used in ACORD are very similar with respect to their structure, their constraints for anaphora, and behave identically with respect to non-linguistic information required, it seemed the most efficient solution, which removed the necessity for parallel development of RESOLVER s for each language, to construct a special module that takes care of the complete process of resolution for all three languages and that

queries all the components that have the required information. To deal with differences among the languages it was required that the module be parameterised to the language, or can be controlled by special features in the information from the parsers.

To achieve this, several problems had to be overcome. The resolution process needs morpho-syntactic information, eg. about gender, precedence and grammatical hierarchy, which somehow had to be made accessible. Since in the project two different grammar formalisms were used, LFG for German, UCG for English and French, the RESOLVER could not also make assumptions about the kind of information available from the parsing process, or about the way the information is represented at the end of parsing. Therefore it was necessary, to develop a common representation for the required information in a form independent from the grammar formalism used, to pass the required information. The solution developed was to associate with the InL terms *occurrence information*, which was integrated into the InL used as common semantic representation language. The RESOLVER can extract this information from the incoming formula, use it for resolution, and then deliver a cleaned up resolved InL to the Knowledge Base for further processing. [1] This architecture also required changes in the parsers which then had to provide the additional occurrence information for the RESOLVER.

The development of the RESOLVER was guided by the acknowledged principles and constraints from syntax and semantics, and took up ideas about discourse structure and focussing, which have evolved during the last decade. Especially, we took up the syntactic principles from GB-Theory (Reinhart 83), semantic constraints on accessibility of antecedents from DRT (Kamp 83), and made use of notions like *current focus stack* and *focus space* introduced and developed by Grosz 77, Sidner79, Grosz and Sidner 86. The guiding principles and strategies used have been described in Deliverable T1.7'(a). The extensions described in Deliverable T1.7'(b) have only partially be realized but were not included in the system, since other components did not support the extensions.

In addition to the resolution facilities the RESOLVER supports disambiguation, especially of adjuncts. This is done in the same way as the plausibility check. The modular structure makes it easy to extend the coverage to other phenomena of contextual reference. An extension to deal with tenses and temporal reference was started but was not completed (Eberle and Kasper 89, Deliverable T1.7'(b)), since the KB cannot deal with temporal information. Also, the treatment of plural anaphora, as described in Deliverable T1.10 could only be partially implemented because the KB did not have the required functionality for a fuller treatment. Deliverable T1.7'(b) and Deliverable T1.10 discuss further possible extensions as to discourse particles and VP-ellipsis which are not yet treated.

The RESOLVER needs interfaces to other components of the system:

- to the graphics system, to resolve demonstratives

[1] In this section the term KB includes the TP since from the point of view of the RESOLVER KB (in a narrower sense) and TP form a unity.

- to the KB, for identifying the referents of definite descriptions and checking the conceptual relations of conceptual subsumption and possession, between objects. Also, the plausibility check and adjunct disambiguation need access to the frames of the conceptual KB.

In the first phase of developing the RESOLVER only purely linguistic knowledge was used to constrain the resolution process. Especially in the last year, work concentrated on integrating non linguistic knowledge from the KB into the RESOLVER by developing the required interfaces to the KB for resolving complex definite descriptions like *the truck in Paris* and using it for plausibility checking, and disambiguation of adjuncts. This originally was expected to be done by the KB itself. The current solution seems to be more efficient from the point of view of architecture, since it reduces the necessity to backtrack from the KB process to the resolution process in the DM.

The module is in some respects ACORD specific, depending on the special properties of representations and occurrence information used in ACORD since it uses Prolog unification and matching over the structures. But the techniques, architecture and procedures should be readily adaptable to other kinds of representation and applications. It is a general, domain independent tool, since all the extra-linguistic and domain specific information required has to be provided by external knowledge bases. Its modular structure allows to extend its coverage without requiring major revisions on the existing modules.

7.2 Technical Details

7.2.1 Software and Hardware Requirements

The RESOLVER is implemented in PROLOG, and can be used with various dialects of that language. The storage requirements are as follows (approximate figures for Prolog+) :

Disk storage	75 KB
Heap	113 KB
Atom Space	41 KB

Further space is required for the storage of dialogue history.

7.2.2 Files

The files for the RESOLVER can be found in `$ACORD_STUT/resol`. The `Readme` file there gives some general information about their contents. The file

`sys_resol.cpr`

is the load file for the resolution software. It defines which files have to reconsulted for the ACORD-system. The files also contain very detailed documentation of nearly all the predicates defined there, and of the ideas underlying the procedures. The files `interface` and `consalone` are needed only when the RESOLVER is to be employed without the DM of the complete system, and have to be adapted to the special environment or configuration of modules used. At present, `interface` defines top level predicates to run the RESOLVER with the German parser, while `consalone` is used to interface the conceptual KB of ACORD for plausibility checking and disambiguation without the DM.

7.2.3 Functionalities

The resolution software implements or supports the following functionalities of the ACORD system:

- resolution of anaphoric and cataphoric uses of personal, reflexive and possessive pronouns
- resolution of definite descriptions and demonstratives
- building and updating the dialogue history
- disambiguation of ambigous adjuncts/prepositions
- disambiguation of thematic roles
- plausibility check

7.2.4 Software Description: General Design

The RESOLVER runs through the incoming InL from the parsers and searches for terms and their occurrence information which have to be resolved. When it finds one, it activates special modules to deal with the special kind of pronoun or referring expression. The following modules are currently provided

- personal pronouns
- reflexive pronouns
- possessive pronouns
- definite descriptions
- demonstratives

Besides these modules the RESOLVER consists basically of three (kinds of) routines:

- search routines for antecedents which respect the accessibility constraints of DRT, and make use of heuristics like parallelism and assumptions about discourse structure.

- checking routines to check for global syntactic constraints, and for checking the plausibility of readings obtained against the frames of the conceptual KB.
- update routines which remove the occurrence information from InL and construct the antecedent stacks for the following discourse, also guided by assumptions about discourse structure.

Successful resolution usually leads to the unification of the pronominal term with the antecedent term. In case of definite descriptions the RESOLVER can also add semantic material to the InL, expressing other kinds of conceptual relations between objects than identity.

The RESOLVER at present covers only nominal anaphoric reference to objects, and does not deal with temporal or propositional anaphora. Another basic restriction is that it can only resolve antecedents which have been explicitly introduced in the discourse, that is, it cannot 'construct' new antecedents, as often would be necessary to handle plural anaphora (Deliverable T1.10).

The RESOLVER fails if it does not find accessible antecedents for all pronouns satisfying the global syntactic constraints. Its behaviour is a bit different when the plausibility check fails, which presupposes that possible antecedents were found. If no 'plausible' reading can be found, the RESOLVER does not fail but just passes on the first reading it found to the KB as the intended one.

The RESOLVER allows personal and possessive pronouns, usually only to refer back to antecedents on the current focus stack (or in the sentence itself), which includes only objects mentioned in the immediately preceding utterance. That means that the user usually cannot refer back by pronouns to elements introduced two or more sentences before. The only exception is when the RESOLVER can interpret the utterance as a resumption following an elaboration, since the resumption will activate the focus stack in use again before the elaboration is entered. In all other cases, the user is expected to use definite descriptions for 'long distance anaphora'.

7.3 Relations to Other ACORD Components

Call The RESOLVER is called by the predicates

`dm_nl_resolve(Unresolved,Resolved,Stack)` or

`dm_nl_resolve(Unresolved,Resolved,Stack,Strategy).`

`Unresolved` is the unresolved InL coming from the parser, while `Resolved` contains the resolved InL without any occ-terms and without disjunctions. `Stack` contains the update for the dialogue history. The second predicate for calling the RESOLVER additionally returns the discourse function of the current sentence which influences the resolution

strategy which the RESOLVER employs in `Strategy`. `Unresolved` must be instantiated, while the other variables should be uninstantiated, when calling the predicate. The resolved InL can still contain uninstantiated variables as identifiers for discourse referents (that is the last argument of the `terms`). Before passing the formula on to any other component, and before storing the update of the dialogue history (see below), they must get instantiated. It is not important how they are instantiated, since the RESOLVER does not care about the internal structure of the identifiers, but of course different objects must have different identifiers. In the ACORD system the instantiation is done by the DM according to the format the KB expects. For using the RESOLVER separately from the system, the RESOLVER supplies the predicate `instantiate(Resolved)` which instantiates variable identifiers with integers.

Dialogue history The variable `Stack` contains the update for the dialogue history. For storing this information, that is, for doing the update, there exists the predicate `dm_nl_assert(Stack)`. The `Stack` is a list containing two sublists, the one representing the current focus stack, which is used for the resolution of pronouns, the other list containing the update for the discourse universe. The current focus stack stores the occurrence information, while the discourse universe contains only the `terms` together with the number of the sentence in which they occurred. Only the last occurrence is stored. The current focus stack is stored under the key `'#curr'` in PROLOG's secondary database, after erasing the old focus stack. The elements of the discourse universe used for resolving definite descriptions, are stored under the key `'#antec'` in the secondary database, in the format

```
d(Nr,Term)
```

where `Nr` is the number of the sentence in the discourse, and `Term` an InL-discourse marker.

The stacks do not necessarily store all the terms (and their occurrence information) introduced in a sentence, but only those which are regarded as potential antecedents for pronouns in the subsequent dialogue, that is, only those which are regarded as accessible in the sense of DRT.

Interfaces and externally defined predicates

To access the KB and the graphics system (for resolving demonstratives) the following predicates have to be supplied from outside:

```
dm_who_is(Term,Description,Answer)
```

`Term` is the object asked for, `Description` is the description the object must satisfy, `Answer` is a list of terms for objects, which satisfy the description. The predicate is equivalent to the query: Which objects satisfy the description?

`qu_cr_verify(hasa(Class1,Class2),Answer)`

`qu_cr_verify(generalize(Class1,Class2),Answer)`

`qu_cr_verify(specialize(Class1,Class2),Answer)`

`Class1` and `Class2` are class names for objects, `Answer` an uninstantiated variable, which gets instantiated to the first argument of the query if it succeeds. The queries concern components etc of things, and conceptual subsumption.

`dm_click_request(N,List)`

asks the graphics system for the objects pointed at by the `Nth` mouse click, which are passed back in `List`.

`dm_check_pred_frame(Pred,Answer)`

`Pred` is an InL-predicate with its arguments, `Answer` is a variable which will get instantiated to `yes` or `no`.

`dm_check_adjunct_frame(Adjunct,Term1,Term2,Answer)`

`Adjunct` is a possible sense of a preposition (an *adjunct function*), `Term1` and `Term2` are the terms for the arguments of the `Adjunct`, `Answer` is a variable which will get instantiated to `yes` or `no`.

`ta_k_cla_suprem(Class1,Class2,Suprem)`

`Class1` and `Class2` are KB class names like *truck*, `Suprem` is a variable which will get instantiated to a superordinated concept which subsumes `Class1` and `Class2` if there is one.

`diag(Mode,Text)`

outputs diagnosis `Text` when the system is run in mode `Mode`.

7.4 Potential of Development

Extensions Because of its highly modular structure, it is not too difficult to extend the coverage and functionality of the RESOLVERWe have eg. started to extend the resolver to tenses and temporal reference. This could not be included in the current system because other components do not support the representation of temporal information.

Possible combinations with other components Since the RESOLVER does not make any assumptions about data structures and implementation of other components of the system, it is a highly portable module that can easily be used also with different systems having a comparable functionality.

For its full functionality the RESOLVER needs access to

Graphics System for demonstratives

Conceptual KB for simple definite descriptions, disambiguation, and plausibility check

Factual KB for complex definite descriptions

With reduced functionality the RESOLVER has been used in the following combinations:

- *Parser + Resolver:* no demonstratives, no correct disambiguation, reduced functionality for definite descriptions
- *Parser + Resolver + Conceptual* KB*:* no demonstratives, no complex definite descriptions

Chapter 8

Dialogue Manager and RPC

Toine van Hoof,
Traude Manz
FhG

This chapter describes the central component of the system which, apart from the classical tasks of low-level and high-level communication, functions as the integration medium for natural language and graphics.

8.1 Scientific Background and Development

8.1.1 General Task Description

One of the most important issues covered by the ACORD project is the combined use of text and graphics in updating and querying of knowledge bases. Aim of the project is to provide the user of the system with an integrated treatment of both modes for input and output. It was decided that all the functionality involved in attaining this flexible, integrative, user interface should be encorporated in one module of the ACORD system, i.e. the Dialogue Manager, DM.

The DM is the central component of the ACORD system. In order to achieve the functionality mentioned above, it has to communicate with all the other components. This does not present any special problems, because the DM already is responsible for inter-module communication, being one of its classical tasks. Because of this, much effort had to be put into the specification and implementation of a whole series of interfaces.

Furthermore, it was decided that the DM is responsible for the proper startup and

shutdown of the complete ACORD prototype system, as well as for the maintenance of low level process communication between the separate system processes.

The following subsections will deal in more detail with the above mentioned issues.

8.1.2 Integration of Text and Graphics

The integration of these two modes brings up several theoretical and technical issues. Foremost is the issue of the semantic of deictic phenomena and natural languages. What does the user *mean* if it uses deictic expressions like *this* and *that* in its natural language input, while it has made a mouse selection or some other graphic manipulation? And how are we to interprete sentences in which the user uses perceptory words, especially the verbs *to see* and *to show*?

We started our investigations with a survey of the literature on already present combined graphics-textual systems. This survey is documented in Deliverable T4.1'(a). In this study it became obvious to us that the systems mentioned in the literature can only partly present a basis for the work in the ACORD project. Further theoretical work was needed, both of a linguistic and a logical kind. This resulted in Deliverables T4.1'(b) and T4.1'(c). The first presents a classification of deixis phenoma. The second tries to abstract semantic principles from natural language use of perceptory words in the theoretical framework of situation semantics.

Of these two topics only the work on deixis was continued in the project. Work focussed on the resolution of deictical expressions. For this purpose a small DM component, the deixis RESOLVER was specified and implemented in the first versions of the ACORD demonstrator. The parsers presented a special marking for deictic expression in the (unresolved) semantic representation they deliver by which the deixis RESOLVER was triggered. Through communication with the graphics system it found out whether a mouse click had occurred and, if not, it prompted the DM to ask for one. The deixis RESOLVER got from the graphics system all the object identifiers of the graphical objects that enclose the selected pixel on the screen. Then it had to check to which semantic objects they correspond and had to choose the correct resolvent. In this choice it made foremostly use of semantic information. E.g. *This truck* already closes down the range of possible candidates enormously because of the class information given. Also the verb of which the deictic term is an argument gives further information to decide upon the likelihood of certain candidates.

This module was integrated in the fourth year with the anaphora RESOLVER built by the linguists. This integrated RESOLVER component is installed as a subcomponent of the present Dialogue Manager.

The inverse of user deixis, deixis generated by the system in responses, is normally called highlighting. Originally an inverse component of the deixis RESOLVER the highlighter, was implemented to highlight those objects on the screen that appeared in a system response to a user query. As with the deixis RESOLVER it has been integrated into a

PLANNER (general inverse resolver) that also takes care of the proper pronominalization of the system's textual response, and is installed as a DM subcomponent as well.

Deliverable T4.3 presents the theoretical aspect of this work, whereas the reader can find the functional specification of the above components in Deliverable T4.7

8.1.3 Visualization of Knowledge Base Information

The central issues in visualization are the questions of *what* one wants to visualise and of *how* one is to interprete user manipulation of this visualization. A study of the literature on these topics and our practical experience in this field gathered in the ESPRIT project LOKI led us to the introduction of a separate visualization module for which we developed visualization principles.

As described in Deliverable T4.7 for the final demonstrator we now visualize for the ACORD application domain (transport business):

- locations (e.g. cities) and connections between locations (e.g. European motorways) summarized in a map,
- vehicles (e.g. trucks and tankers) on that map, and
- transportable goods like printers, screens, cables and disks in bar charts.

For all these objects not only the objects as such, but also some of their properties and relations are part of the visualization. The visualized properties of a location are its geographical coordinates, its name, and a depot indicator.

Locations which have a depot indicated have an additional property: They can be "asked" (through natural language or direct manipulation) to show the contents of their depot in a bar chart. That chart contains one bar for each of the goods mentioned above. The height of the bar gives the amount of the good present in the depot. If the good is not in the depot the height is zero. If the user told the system that the depot contained, e.g. *some printers*, the height of the bar is about 50% of the maximum amount and the number indicator shows a question mark. Thus the user can later on change that value by direct manipulation as well as through natural language.

The visualized properties of a vehicle are its kind, its location, its goal, and its load. The load is presented in a bar chart similar to the one for a depot, only the maximum values and the object it belongs to are different. If a vehicle is in a city which has a depot, the user can unload the vehicle into the depot or load it from there by direct manipulation as well as through natural language. KB and GS have implemented some constraints which maintain the dependency between depot and vehicle charts in such cases.

The visualization component of the DM does the necessary processing on KB and GS-

data (GS: the graphics system) concerning visualized objects and objects which are subject to get visualized. It transforms direct manipulative movements of vehicles into a semantic representation which is non-distinguishable from the according representation of a natural language input. As a further functionality the visualization catches all imperative natural language input and interpretes it as commands, e.g. to show the contents of a depot or the load of a truck.

According to developments within the DM-TP-KB cooperation concerning plurals and quantifiers it is now possible to visualize even some implicit knowledge which was in Deliverable T4.7 considered to be too complex for the visualization component to do. E.g. the user inputs *truck5 carries 5 printers* and *every truck which carries 5 printers is at Berlin* lead now to the visualization of truck5 at Berlin. If the user talks about a set of trucks like *3 trucks* or *some trucks* they are also visualized on the map as soon as their location is known. Only the loads of such trucks are not shown. Bar charts are only created for single trucks like *truck1* or *truck2 and truck3* which mention two single trucks.

8.1.4 Interfacing, High and Low Level Communication

Initial work on the DM focussed upon the practical issues of the software architecture of the ACORD system. In Deliverable T4.4 three architectures are discussed: the single-process model, the multi-process layered model and the multi-process network model. From these three the last one was chosen.

The first implementation of this architecture was with UNIX pipes. The top level of the package was some PROLOG process that had some dynamically linked C functions for forking all the necessary further PROLOG processes and setting up the communication pipes between them. Standard input and output of the processes was redirected to these pipes. In each process the code for the specific modules was loaded and the main loop of the process was started.

The main disadvantages of this setup were that the ACORD demonstrater was confined to a single machine, and that it was impossible to interactively debug the single processes because of the redirection of standard input and output. Whereas the first disadvantage merely made the ACORD demonstrator very slow (much time was spent on swapping the large processes), the second was more serious because it slowed down the integration and testing of the complete system.

To do away with these disadvantages a new implementation was made making use of the facility Remote Procedure Call (RPC). Now initially in a shell script a server process and all its clients (the ACORD modules) are started, that load their code and start their main loop. Every process keeps its standard input and output. The processes can be distributed over several machines.

This work has covered more than what was specified in the work packages of the original ACORD Workplan. Especially the work on the low level communication implementation

through the UNIX facillity RPC was not originally foreseen. The use of the program package developed for this purpose has been very fruitfull for the ACORD project. The package itself has already proved to be easily adaptable to other multi-process architectures.

At the higher level the communication between the modules is defined by the ACORD specific interfaces. The main task of the DM at this level is to catch all the incoming calls in the main loop of a process and to process them in the appropriate way, eventually sending them to other components or processes and routing the answers back.

The Dialogue Manager consists of the following modules:

- input processor
- RESOLVER
- knowledge base / theorem prover interface
- visualization component
- PLANNER
- output processor

The RESOLVER (see 7) and the PLANNER (see 2) are constructed by the linguistic teams. They incorporate the functionality of the former deixis components. The functionality of the other modules is described in the Deliverables T4.4 and T5.10.

In the interface to the KB process that contains the Knowledge Base and the Theorem Prover is some extra functionality in the preparation of updates and queries. This is especially the case with quantified information (e.g. natural language updates with all quantifiers or plurals). In this sense the DM functions as an extended KB management module.

8.2 Technical Details

8.2.1 Software and Hardware Requirements

The overall startup routine of the system and its tailoring parts are implemented as csh-scripts. Reader, diagnostics-window and low level RPC communication parts of the system are implemented in C. Anything else concerning dialogue managing and visualization is implemented in PROLOG (prolog+). It runs on SUNs (3/50, 3/60, 3/260, 4) with operating systems between SUN OS 3.4 and SUN OS 4.0 as well as on VAXes (e.g. VAX 11/750) running 4.3 BSD UNIX.

The disk storage requirements for RPC and DM are as follows:

source code for RPC	81 kbytes
source code for DM	73 kbytes
compiled code for RPC	284 kbytes for SUN OS 3.4 or 3.5
compiled code for RPC	21 kbytes for SUN OS 4.0
at run-time created startup files	5 kbytes on /tmp

The difference in the compiled codes comes from the fact that SUN OS 4.0 loads system libraries dynamically whereas SUN OS 3.4 and 3.5 don't.

In order to run the communications only on a SUN 3 without distributing processes and without the code for the DM or any other module except RPC loaded you need about 11 MB free swap space. Running the whole ACORD system requires 22 MB free swap space.

For SUN OS 3.4 the maximal number of charts is restricted to 3 according to a file descriptor limit, for SUN OS 4.0 this limitation no longer holds.

8.2.2 Software Organization

	ACORD_HOME	stands for the root of the ACORD-software hierarchy,
Assuming that	ACORD_RPC	stands for ACORD_HOME/acord/rpc and
	ACORD_FHG	stands for ACORD_HOME/acord/fhg

the files for startup of the ACORD system, for RPC and DM are located as follows:

Startup files When logging in as user *acord*, or as an other user you should source the file `.acord_config` on `ACORD_HOME` first. Then your command path has the additional path `ACORD_HOME/bin`. There you find the startup command file `acord`. For distributing processes on different hosts you need a file `mymachine.conf` (replace `mymachine` by the name of the machine on which you start the system) on the directory on which you start the ACORD system. An example for it is the file `ACORD_RPC/host.conf`. The defaults assumed for all parameters that you don't specify there are given in the file `ACORD_RPC/rpc.conf`. Further information about tailoring the ACORD startup can be found in the file `ACORD_RPC/README`. Additionally this file gives more details about how to run processes on a VAX and about how to use the RPC module including diagnostics and reader.

For a first configuration of the ACORD system the files `ACORD_HOME/Makefile` and `ACORD_HOME/mk.acord_config` are needed as well. The file `ACORD_HOME/Readme` gives more details about a first installation of the whole ACORD system.

RPC code The code for the communication tasks is located in the directory `ACORD_RPC` and its subdirectories. Files with the extension `.cpr` contain PROLOG code as well as the PROLOG init file `ACORD_RPC/.prologrc`. The files starting with the prefix `main_` contain the main loop definitions for their processes as well as the commands to load

all software belonging to their processes. The file **utils.cpr** provides some predicates which are useful for all processes and modules.

The C code is to be found in **ACORD_RPC/c**. The compiled C code will be put into the directory **ACORD_RPC/SUN_ACORD**, where **SUN_ACORD** stands for a combination of "SUN3" or "SUN4" with "OS34" or "OS40" depending on the SUN and the SUN operating system used. When distributing the processes on several machines it is possible to have different SUN's with different operating systems working together, provided the C code is compiled for each of those machines. **SUN_ACORD** is an environment variable which is automatically defined by the shell script **ACORD_HOME/.acord_config**.

DM code As it is not part of the **main**-files mentioned above, it is put into the directory **ACORD_FHG**. The first two letters of the file names there indicate the process in which they are loaded. The main part of the visualization component is put into the subdirectory **ACORD_FHG/visu**. The visualization load file and the interfaces needed for other components like the RESOLVER and the PLANNER (inverse RESOLVER) are put into the DM directory **ACORD_FHG**.

8.2.3 Software Description: General Design

Startup phase The general idea for the startup of the ACORD system is that a user should only know the full path for the command that performs an automatical startup of the system. Anything else will either be done automatically during the startup or when installing the system on that machine.

The reader It reads the user's input and sends it to the dialogue manager for further processing. System answers are sent to it for printing. Since it uses a text subwindow, scrolling is possible as well as editing the text. For input to the system make sure that you type your input after the last prompt from the system. Otherwise your input is ignored. A return character sends all characters after the last user prompt to the DM. The characters appear then in inverse video and are thus selected for text subwindow functions like "put" and "get".

The diagnostics window It contains one text subwindow for each of the ACORD processes. Depending on the diagnostics mode it displays extended debugging or simple demonstration messages. It is used to show, what is going on within the different modules during processing a natural language or direct manipulation input. It also shows the flow of the communication between the different modules. All data appearing in demonstration mode appear also in the extended debugging mode, where additional information is presented. How much additional information is there depends on the use of the predicates **diag/2** and **diag_pp/2** which are described below. As shown in the

use of the GS it is possible to send messages to the diagnostics window from C code as well as from PROLOG code.

The server The server **svc** registers the different ACORD processes and receives messages from them. It stores them until the addressee signals that it is listening. Then the server sends the message to the addressee and afterwards continues waiting, storing and sending messages. The use of symbolic names for the different ACORD processes and the registration mechanism allow restarts of single processes without restarting the whole system. The signal for listening always contains two masks for the server, the first one indicating which messages are expected and the second one indicating which messages the server should ignore. All messages not covered by one of the two masks remain in the server's storage until they are requested or until the server stops.

The ACORD processes DM, GS, KB and DM The RPC code starts all these processes, performs the registration at the server and starts the main loops. Such a main loop consists of an initializatin phase where e.g. the map data for the initialization of the GS are created and the real loop, where the process reads messages from the server, processes them, sends answers back and starts waiting for messages again. Before starting the main loops the software for the different processes is loaded using a predicate `sys_load_from(Directorypath, File)`. If a process has been started as dummy process, the predicate `sys_load_from/2` will do nothing, i.e. the module code is not loaded.

Dialogue managing The central part of dialogue managing is done in the process DM. Natural language input from the reader is received there and sent to the NL process for parsing. After parsing the InL comes back to the DM, where it is now processed, firstly by the RESOLVER, and secondly by the DM itself and by the PLANNER. If the input is a command, it is passed along to the visualization component for interpretation, otherwise the KB gets it for storage or retrieval. In the case of a question the PLANNER is considered again for providing code to the natural language generators. If the KB stores new information it also returns the indicators for the visualization component to update the GS output.

For further details on dialogue managing see the Deliverables and the commented code mentioned above.

8.2.4 Illustrative Examples

Examples concerning dialogue managing and visualization can be seen best in the running system. We therefore present here two examples on startup tailoring, which is usually not shown in demonstrations.

Assume that we want to start the ACORD system on a SUN workstation called `fix`

and we want to distribute some processes on machines named **prefix** and **suffix**. Let us further assume that the code of the ACORD system resides under the directory **ACORD_HOME**, that our own home directory is **MY_HOME** and that we are running a csh or tcsh.

First and once for all times we copy the file **ACORD_HOME/.acord_config** to **MY_HOME** and put a source command for it into our file **MY_HOME/.cshrc**.

Now we create a file **fix.conf** on the directory on which we want to start ACORD . After calling **acord** the system automatically starts up.

Here is an example of what our file **fix.conf** looks like:

```
#! /bin/csh
#

set RPCkb       = "rsh suffix"  #   i.e. "rsh <hostname>" instead of ""
set RPCdm       = ""            #   Subprocess will start on this host.
set RPCnl       = "rsh prefix"  #

set RPCDgs      = ""    #      "dummy"    #    only for testing purpose
set RPCDdm      = ""    #      "dummy"
set RPCDkb      = ""    #      "dummy"
set RPCDnl      = ""    #      "dummy"

set RPCmode =  "sun"    #      "nosun"  means tty version with dummy gs

#       Diagnostics mode one of (x,d,n)

set DiagMode = "d"      # n means no diagnostics at all, x is debugging

#-------- ------------------------------------------------------------#
```

If we had not created that file, the system would have started all the processes on the machine on which we started it, assuming it to be a SUN and the diagnostics mode to be the demo mode "d".

8.3 Relations to Other ACORD Components

8.3.1 Interfaces

All the interfaces described here need the RPC and DM software for a welldefined functionality if not stated otherwise. Thus only the files containing the interface predicate definitions are given below.

Interface between DM and RESOLVER . The RESOLVER needs access to the objects which correspond to deictic expressions in the natural language input. It also uses some additional information about those objects concerning their movability on the map. The interface is defined in the file `ACORD_FHG/dm_selections.cpr`.

`dm_click_request(N,List)` always succeeds. There can be several deictic expressions in one sentence. They are enumerated from left to right within the sentence, assuming that the user will point on the graphical objects in the same sequence as it uses the deictic expressions in its natural language input. `N` is the number of that deictic expression the RESOLVER wants to know about now. `List` will be instantiated by this predicate. It will be empty, if nothing was selected. Otherwise it will contain elements of the form
`movable(obj(_id,_sort,_class,_inst))` and
`nonmovable(obj(_id,_sort,_class,_inst))`.
The system will try to prompt the user for a selection if it is not already available. The empty list can only occur if the GS is not loaded.

Interfaces between DM and PLANNER The PLANNER (inverse RESOLVER) needs some interface predicates to decide whether the text output to be generated will include deictic references or not. It also needs access to the knowledge base for alternative object descriptions.
The interface definition is given in the file `ACORD_FHG/dm_planner_interface.cpr`.

`dm_planner_visible( Term )` succeeds if `Term` is visible and fails otherwise. `Term` is of the form `term(A,B,C,D,Class,Name_or_ObjTerm)`.

`dm_planner_highlight( Term )` succeeds if the `Term` was highlighted on the screen and fails otherwise. `Term` is of the form `term(A,B,C,D,Class,Name_or_ObjTerm)`.

`ta_make_unique_descr( X,Y )` is a predicate call which is passed on to the KB process. Its result is passed back to the DM process and it always succeeds. `X` must be instantiated and `Y` must be a variable, which will be instantiated by the KB. The semantics of this predicate are defined in the KB.

Interface between DM and GS Most parts of the interaction between DM and GS are done by calling interface predicates from the GS. The only exception are direct manipulations. There the GS sends a message to the RPC part of the system, which then simulates a call from the DM to the GS. The interface for this message is a C-function called by the screen-manager of the GS. Its code resides among the GS-C-code in the file `ACORD_HOME/acord/edc/Interp/C/kgs_event_msgs.c`.

`int kgs_clnt_dmei_register()` must be called when starting the GS-screen-manager. It registers the screen-manager in the RPC-server and returns "1".

`int kgs_clnt_dmei_unregister()` is called when stopping the screen-manager to unregister it in the RPC-server. It also returns "1".

`int kgs_clnt_dmei_send(request)` sends a char string to the RPC-server. The call used in the GS-screen-manager causes the server to simulate the above mentioned DM-call to the GS for direct manipulations. The `request` is declared as `char *request;`. The function returns "1" on success and "0" on failure of the message sending.

Interface between DM and KB The KB doesn't call any predicates of the DM, it is the DM which calls interface predicates of the KB.

Interfaces between DM and NLparsers and generators What is stated above for the KB holds for the NL (natural language) components as well.

Interfaces for diagnostics and debugging All components of the ACORD system have access to the interface predicates for passing messages to the diagnostics window. They are loaded with the RPC code.

`diag(Flag, Form)` prints the PROLOG expression `Form` into the diagnostics window. It automatically chooses the subwindow of the process in which it is called and preceeds the form with process name and an increasing message number. `Flag` is either `x` or `d` meaning "extended debugging mode" and "demo mode" respectively. `Form` must *not* be a variable. The predicate always succeeds.

`diag_pp(Flag, Form)` does a pretty printing and is recommended for printing semantic representations like InLs. It does not preceed a number or process indicator, but it also prints in the diagnostic subwindow of the process in which it is called. If `Form` is a variable, `a(Form)` is pretty printed. The predicate always succeeds.

`sys_loop` restarts the main loop of the process it is called in including all initializing parts of that loop. E.g. in the DM process it would create the map for the GS again and it would check for the languages loaded in the NL process again before starting the real loop. This predicate is here for historical reason. Better use one of the predicates below for restarting loops.

`dm_loop`, `gs_loop`, `nl_loop` and `kb_loop` restart the loops of their processes without the initializing parts mentioned for `sys_loop`. Use these after aborting a loop for debugging.

The file `ACORD_RPC/utils.cpr` is loaded with the RPC code and contains some useful predicates for testing memberships, appending lists, concatenating and splicing symbols, initializing, increasing and decreasing counters, ... For further details and exact interfaces of these predicates consult the source file mentioned.

8.3.2 Dependency Constraints

Without RPC no communication between processes takes place, i.e. all there is no communication between KB, GS, DM and the natural language process without RPC. They all depend on it.

The DM needs input from a parser, it needs the RESOLVER subcomponent and the KB, which reacts on the DM's demands. Without all the system components a proper work or testing of the DM is impossible.

8.4 Potential of Development

8.4.1 Use in other situations

The "extended knowledge base manager" (DM code) contains parts of code which are used whenever conversion from an InL-representation into the KB's semantic representation is needed.

The visualization subcomponent of the DM is no independent tool either. But it is easier to adapt to new interfaces because they are more restricted. Extensions in the functionality like, e.g. pointing on user actions and enabling the user to program the interpretation of those actions through natural language, require further research.

8.4.2 Implementation in Other Systems

The DM code is highly system dependent. Implementation in other systems is only possible if they use the same kinds of representation as ACORD does. As the code is designed around the mechanism of term unification provided by PROLOG, it is likely that reimplementaion in some other language would prove time-consuming.

With some effort it were possible to adjust the visualization component to, e.g. an other graphics system or another knowledge base, as long as the description of the objects as "obj(Id, Sort, Class, Name)" remains. Again reimplementation in some other language could prove time consuming.

The inter-process communication software, the reader and the diagnostics window (RPC code) are good candidates for tailoring and implementation in other systems. They all can be used independently. If all the new processes were C processes it could help to reimplement the PROLOG parts of the communication software in C, but it is easier to change the interfaces if it remains in PROLOG.

8.4.3 Use with Other Components of the System

The startup code allows the switching on and off of the several system components. This can be useful when testing only some modules with running inter-process communication. Restarts of single processes are possible. The whole system can also be run without the graphics, e.g. for testing only the natural language parts or in case there is no SUN available to run the graphics system. Without RPC- and DM-code no combination of other system componentes is possible, except possibly combining natural language parsers and generators.

Chapter 9

Graphics System

John Lee
EdCAAD

This chapter describes the Graphics System which allows deictic pointing by the user, highlighting of objects on a map and direct manipulation.

9.1 Scientific Background and Development

9.1.1 Introduction

EDC came to the ACORD project with experience of CAD, and our research interests have always been concerned with designers' use of computers and their use of drawings as a mode of expression. The objectives of EdCAAD's contribution to the project, as defined in the original version of the *Technical Annex*, were twofold: firstly, to develop a general graphical production environment based on a binding of the newly-emerging computer graphics standard "GKS" (Graphical Kernel System — Enderle et al. 84) to PROLOG; and secondly, to develop further the interest in general graphical communication that was beginning to emerge at EdCAAD. Accordingly, we were not scheduled to start work on our "theoretical" tasks until the second year of the project, after the GKS development had been implemented, although the early Deliverable T3.2 provided an overview of the field of interactive graphics and defined our outlook on the issues. An observation made in this deliverable was that literature describing theoretical work on computer graphics systems that interact with knowledge bases is somewhat thin. Thus, unlike the linguists, we did not have at our disposal at the start of the project a sound body of theoretical knowledge which could be used as the basis of an implementation and we knew that a large part of our subsequent work would be the identification of a suitable such framework.

A central ambition of the project as a whole was the integration of all components. This meant an investigation of how alternative modes of expression such as text and graphics could be seen to be similar, and to what extent the same formal techniques could be used to represent expressions in either medium. To this end, our intention was to push the analogy of graphics as language as far as possible. Considerable effort was devoted to investigating the extent to which available semantic representations such as DR-theory (Kamp 81), and notational variants thereof such as InL (Zeevat 86), could be used for the representation of depictions, since the objective was to have a common semantic representation language for the whole of the ACORD system (some of these issues are discussed in a later report, Deliverable T3.5) Several lines of thought on possible kinds of graphical representation were followed up; the results of these appear in deliverable reports (as discussed below) and in a paper describing the principles of the ACORD demonstrator Graphics System (Lee, Kemp and Manz 89), which is further detailed below.

9.1.2 GKS/PROLOG

A great deal of effort was expended in the implementation of what was thought at the time to be a suitable graphical production environment, taking up a substantial chunk of our contribution to the first two years of the project. The result is documented in the Deliverables T3.1 and T3.3. It became obvious during the course of the early part of the project that although GKS may well have been a good choice of graphical production environment when using *vector-based* terminals, the trend was moving more and more towards the use of *raster-based* workstations with their own window systems. In this respect, the work being done on GKS rapidly became outdated, and was superseded by other systems. Furthermore, and more importantly, it was never our aim on the project to develop more sophisticated graphical displays. Rather, the emphasis of the project as a whole was increasingly on *dialogue*, both textual and graphical. The general aim of the project was to allow users to change the knowledge base (and hence the graphical depiction of the knowledge base or parts of it) through either of the two input streams. Thus we would observe that the amount of effort invested in developing GKS/PROLOG in relation to the ambitions of the project was much too great. This effort could have been diverted to the more profitable areas of connecting the graphics interface with the knowledge base, and the design of an appropriate representation strategy. Graphics in general is very sensitive to hardware changes, and we have learned that it is not easy successfully to anticipate the speed of advances in the state of this art, even when using a graphical environment that has recently become recognised as an "international standard".

9.1.3 The First Demonstrator Graphics System

Shortly after it became clear that GKS was a dead end, CEC pressure made it an important goal of the project to produce an interim demonstrator. This was unforeseen in the original *Technical Annex* and precipitated the urgent need for a graphics system

(GS). Since EdCAAD had recently built an extended version of C-PROLOG by adding a dynamic loading capability (Tweed 85), it was decided to use this version in conjunction with special-purpose functions written in C for low-level support, bearing in mind the large programming load that these would entail. At the time, the likely application domain appeared to be some form of financial analysis, so there was heavy emphasis on graphs and charts. The specification of the resulting GS was an internal deliverable of the project (Lee and Szalapaj 86), based on an earlier position paper (Lee, Bijl and Szalapaj 86), but substantial parts of it also appear in Deliverable T3.6(a).

Implemented in time for the 1987 Review, this first version of the GS had extensive functionality (much more, in fact, than the rest of the ACORD system was ever able to address), including a considerable range of general and more specialised C functions for graphical production and interaction (Lee 87). Its interface with the DM, however, was unwieldy and increasingly unlike the evolving forms of InL in the Natural Language (NL) subsystems. This, in part, along with our being obliged to use archaic BitGraph hardware, and a generally heavy emphasis on NL in these early days, led to the GS not being properly integrated into the demonstrator. This had not happened even with an improved version ported to the Sun workstation by the Review of 1988.

It was clear all along that a more semantically principled basis for the design of the system was required. It remains our view that the demand for a demonstrator, in advance of our being able to develop a reasonable theory to support it, was damagingly premature and effectively held back fundamental graphics research already retarded by the time spent on GKS. Our ideas about what such a principled basis should be like led in several different directions, none of which we could afford to ignore.

9.1.4 Graphics and Semantics

As has already been mentioned, we were faced with the problem of identifying a paradigm within which we could work. Without such a paradigm, we could not immerse ourselves in what Kuhn 62 has called "puzzle-solving research". We began by differentiating between two distinct areas of research in graphics. Fistly, computerised picture production, concerned with the design of graphical software systems which concentrate on graphical output, with facilities for user-input to control output. GKS (Enderle et al. 84), and Postscript (Adobe Systems Inc. 87) are two examples of such systems. Secondly, graphical communication, namely the use of graphics to access and modify non-graphical information in a knowledge base. The latter approach, with which we were predominantly concerned, required a general understanding of graphics as language. Most of the theoretical deliverables were aimed at developing such an understanding.

Deliverable T3.4 indicated that the ability to refer to graphical objects and their parts as logical objects could support the linguistic description of depictions, by allowing representations of non-graphical objects to map onto representations of graphical objects. Similarly, amongst the diversity of issues discussed in the Deliverable T3.5 survey of the field was the suggestion that the semantics of graphical objects could be described as formulae in some logical language. To achieve this within any formal system, draw-

ings would have to be classified into types known to the system. Concurrent work in the linguistics stream showed the distinctions to be drawn between graphical sorts and graphical types; these issues provided some of the basis for the notion of semantics elaborated in Deliverable T3.6(a).

Zeevat 86 had suggested that sorts could be arranged in a hierarchy with inheritance between sortal classes, and that the feature list for sorts can be expressed as *dags*. This related to the fact that in the first-version GS the representation of a drawing was encoded as an instanced *dag* structure and also coincided favourably with ongoing work at EdCAAD concerned with developing a representation language for the description of design objects (Tweed and Bijl 88).

This gave rise to a distinctive idea of graphical objects as types which was developed further in Deliverable T3.6(b) and involved much emphasis on the *construction* of new (user-defined) types from existing (primitive) types. The conclusion of this deliverable was that type theory allows the construction of combined text/graphics systems without recourse into separate textual and graphical streams. It thus allows one to take a unified view of such systems.

9.1.5 The Final Demonstrator Graphics System

Many aspects of these investigations were, however, too abstract to be immediately relevant to the demonstrator implementation. Moreover, they dwelt primarily on questions concerning semantics for graphical *objects* — pictures — but said comparatively little about *interaction*. The implementation needed a new account of how this could be interpreted.

A solution was evolved, and described by Lee and Kemp in a 1988 report, later expanded in collaboration with FhG to produce a greater emphasis on dialogue management (Lee, Kemp and Manz 89). As described above, the approach treats streams of events from user interaction analogously to the treatment of streams of words — sentences — in NL. This facilitates the production of a highly InL-like semantic representation which can very easily be handled by the DM.

Implementation of this new solution — which involved the *total* rewriting of the demonstrator GS, though with less general functionality — was in a usable state by the Review of February 1989, and has subsequently been improved considerably, especially in respect of the visual feedback supplied to the user during interaction.

An important intention is that depiction should not be dependent upon any anticipation of graphical objects within the representation of domain objects. Our aim is to define kinds of graphical objects independently from the representation of domain objects. This would allow the subsequent depiction of domain objects without making any changes to domain object representations. Although the domain of the ACORD project is now that of transportation, there is no reason in principle why this domain cannot be replaced by some other domain. This strategy is also applicable within domains. For

example, within the domain of transportation, the type of goods carried by trucks (wine, computers, etc.) may affect the kinds of graphical movements that can be applied to depictions of trucks. We would not want each possibility to be treated as a special case. Rather, we would prefer to express the domain-dependent changes of graphical possibilities in terms of constraints upon the graphical descriptions of particular graphical objects. Graphical representations, including the ability to express such constraints are within the scope of the GS. Actual instantiations of constraints for particular domains are within the scope of the DM. Given any domain shift, one can envisage the preservation of underlying domain-independent graphics operations, which can be redescribed in terms of the semantics of the new domain.

An implication of this view of GS/DM integration is the establishment of an additional link between the DM and the GS templates and rule base, with which the DM could dynamically describe domain-dependent constraints. From the DM point of view, the DM's access to the templates and rule base obviates the need for a static hard-wired visualisation/correlation component for each and every domain to which the system is applied. One can envisage therefore, graphical depictions having particular interpretations in relation to underlying domain-dependent constraints upon graphical representations. These constraints may change in correspondence to a shift in domain, with no corresponding change in the depiction. The same depiction may refer to one or more knowledge domains. Some of the ramifications of these ideas are discussed in Lee, Kemp and Manz 89, as well as in all the later deliverable reports.

9.1.6 Conclusion

It is evident that during the course of the project positive inroads have been made into the problems of semantic integration of graphics systems, natural language systems, and knowledge bases. We observed that any graphics system capable of depicting non-graphical information, has to have some mechanism for the representation of structured objects (e.g. *dags*). The use of such a system for communicative purposes entails the association of rules and constraints with these structured objects. The demonstrator GS exhibits a successful and promising approach to processing such rules.

9.2 Technical Description

9.2.1 Software and Hardware Requirements

As described below, the GS is implemented partly in Prolog (the "Graphics Manager") and partly in C (the "Screen Manager"). The source code for these modules is kept separately, and requires different procedures for making a runnable system. These are also described below.

The graphics manager consists of about 2400 lines of PROLOG code, which will load

and run satisfactorily in C-Prolog with default stack sizes. A considerable amount of C object code will be loaded in, however, as described in the section on the screen manager.

The GS has about 7500 lines of C code in its basic form, with an additional 1500 being automatically generated as described below. The screen manager program which runs as a separate process has more than 600,000 bytes of object code, which rises to over 900,000 bytes in the executable file as compiled and linked using `cc` under SUN OS 3.4 (i.e. including relevant SunView library code); the files dynamically loaded into PROLOG account for another 44,000 bytes of object code. The complete GS, including both PROLOG and C sources, objects and binaries, requires just under 2.5 megabytes of disc space. It requires a Sun Workstation providing the SunView window manager and ideally SUNOS 4.0 or later.

9.2.2 Files

The code for all PROLOG parts of the graphics system is located in the directory `acord/edc/Interp`, with the source code for both the independent screen-manager program and the C-functions that are loaded into PROLOG being in `acord/edc/Interp/C`. Conventionally, files with the extension `.pl` are PROLOG source files, while those with `.c` contain C sources. A number of files are generated automatically when the system is made, and these will be described below. In this document, we adopt the convention of specifying a PROLOG predicate by giving its name and arity (number of arguments), separated by a "/".

All C source-code is in the directory `acord/edc/Interp/C`, although when the system is made several C-files are created and placed in a sub-directory specific to the machine-architecture and operating-system configuration in use — identified by the ACORD environment-variable `$SUN_ACORD` as `SUNxOSyy` where `x` is 3 or 4 and `yy` is 34 or 40 — which is also where all objects and binaries appear.

9.2.3 Software Description: General Design

Implementation Strategy

The overall design of the ACORD graphics system (GS) derives from a desire to have it behave in the system as an independent input module, very like the NL parsers, and to communicate with the dialogue manager (DM) in InL (or something very like it).

These two principal requirements for the behaviour of the GS give rise to a view of graphical interaction as something that can be initiated either by the DM or the user, in either case involving most typically some kind of update to the screen display. This leads naturally to an idea of the display as a constant (although of course not static) structure

in the centre of the module, forming the focus for input from either end. In this context, the term 'display' really denotes the GS knowledge-base structure underlying the actual screen picture, which contains information about what graphical structures are where, what they depict, and what possible changes may be appropriate to them. The input has to be interpreted with respect to this structure, and might well entail changes to the structure as a result. We also have a view of the representation as being strongly semantic, in that it is based on the intended 'meaning' of the graphical depiction, and as including constraints and rules specifying permissible alterations to the display, and their interpretations.

Hence, 'editing' actions by the user are interpreted in the context of the representation, to a level at which they can be assigned a semantic structure encoded in something like InL. Conversely, the DM can send this InL-like material to the GS, which then determines its graphical effect in terms of edits to the display. This approach is particularly clearly shown in the treatment of simple examples like

truck1 goes to Stuttgart

where, given the meaning of the picture as a map where an icon at a node represents a truck at a city, an obvious graphical edit (moving an icon) has an obvious interpretation (moving a truck). We want the InL-like representation of this example to be related appropriately to the graphical update whether the latter is originated by the user, or the former by the DM.

A design based around a parser has evolved from these considerations. The parser uses a 'grammar' consisting of rules which can be used either to accept or to generate InL-like expressions occurring in the head of each rule. The body of the rule contains 'tokens' (created from user-events), 'actions' (usually specifying changes to the GS knowledge-base) and 'constraints' (which have to be satisfied for the actions to be allowed). In accepting input from the DM, if an incoming expression matches a rule, and the constraints are satisfied, then the action is performed (the tokens being ignored). In generating output for the DM, when a substring of tokenised editing actions from the user matches the tokens in the body of a rule, the actions are performed if the constraints allow, and the InL in the head is the output.

Further details about the general motivation and global design considerations of the GS module can be found in Lee et al. 89.

Implementation Tactics

The graphics system itself consists of two subcomponents. One, written in PROLOG, is the graphics-manager, containing the parser and higher-level features of the display representation; the other, written in C, is the screen-manager, which deals with lower-level screen-data representation and the tokenising of user editing-action events. There is also a third subcomponent, known as the DM-shell, which is actually a part of the GS process but which is here treated as being part of the DM: all ACORD modules have a

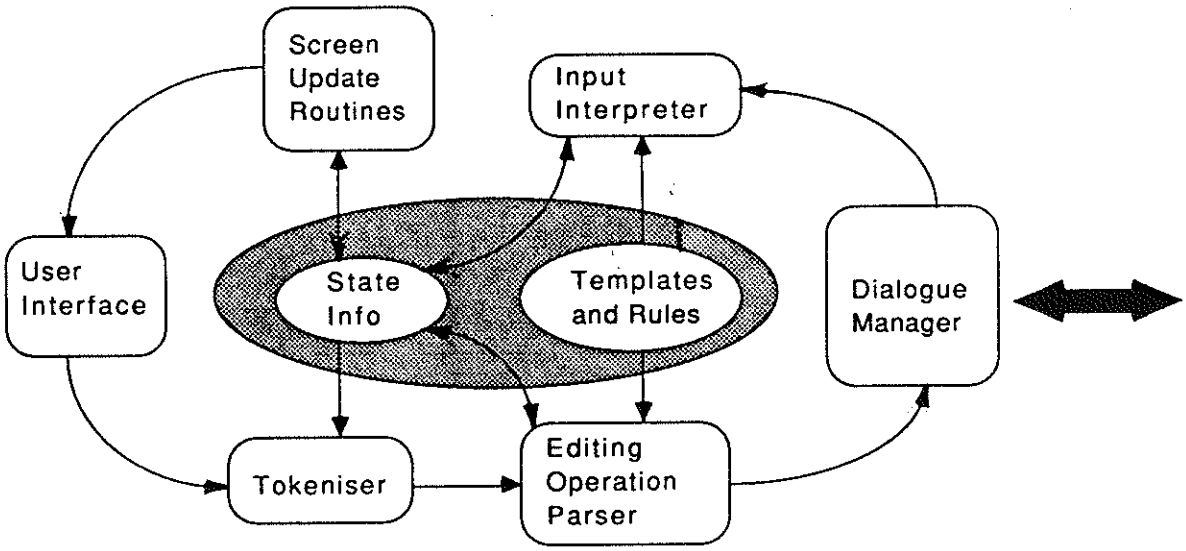


Figure 9.1: ACORD Graphics System Architecture

PROLOG-based DM-shell, interfacing between themselves and the DM proper. The DM-shell of the GS is connected to the DM, as in the cases of the other ACORD modules, using RPC links. The screen-manager, which is an independently executable program, runs in a child process of this PROLOG process, communicating with it via normal UNIX pipes.

Figure 1 shows the logical arrangement of the GS architecture. The GS knowledge-base appears as the central area, and consists of graphical object information with associated rules (stored in the PROLOG component), and graphics state information (in the screen-manager). Around this central area are shown the active processing components which create and update the knowledge-base. The 'editing operations parser' and the 'InL interpreter' are the two aspects of the mechanism which uses the rules in treating input from either the user or the DM; the 'screen update routines' and the 'tokeniser' are both located in the screen-manager.

The Graphics Manager In the theory behind the design of the graphics system, the GS should provide certain primitive graphical objects and functions, out of which more complex ones can be defined. The first responsibility of the graphics manger would then be to accept from the DM a specification of what complex graphical objects there should be, what their interpretations would be, and what editing functions might apply to them (see Lee et al. 89 for a discussion). In fact, this aspect of the design has not yet been implemented. The graphics manager can build objects from a set of primitives, but to the DM it provides a fixed set of graphical objects, with fixed functionality and fixed interpretations. These are the 'graphical object templates'. From the point of view of the GS it should not be a difficult step to implement a feature allowing these

to be downloaded dynamically from the DM, but the DM would need a much more sophisticated visualisation component than it currently has before it could be properly used.

In theory, the templates specify how an object is constructed, and what constraints apply to it. The rules used by the parser/interpreter should also be based on the templates. These features, however, are not as evident in the current implementation even as they were in the old GS, because the graphical domain used in the demonstration scenario (ie, the map with trucks) has not been found complex enough to require them. The graphical objects used are in fact all described as primitives — nodes, links, texts and icons. These primitives can, of course, be parameterised in various ways (eg. position, shape, size and in the case of texts, font), and even the pop-up charts which are used to display information about storage at depots depicted on the map, are really graphics-manager primitives with rather complex parameters.

The job of the graphics manager, then, is to construct an initial picture, given a set of InL-like expressions from the DM, and then wait for further input. (We call our cut-down version of InL the *graphical representation language*, GRL). If an incoming GRL expression makes reference to an object not already depicted, then a new instance of an object of that type is created and inserted in the GS knowledge-base. If full-scale, structured object templates were used, this would be an instance inheriting default attributes from the definition of the template for that class of object; as it is, we simply need an indexed item of the appropriate type. In updating its own information, the graphics manager makes calls to the screen-manager to update the graphics state data and to adjust the screen display where necessary. Input from the user obviously can refer only to objects already on the screen, but a successful parse is likely to result in similar calls to the screen-manager to update the display. Even an unsuccessful parse may require screen-manager calls to control effects such as the dragging of ghost images.

The Screen Manager The screen-manger maintains a set of data-structures which mirror the graphical-object descriptions held by the graphics manager, and describe the details of the screen appearance of an object at any given moment. These structures are set up in response to calls from the graphics manager, as are all changes to them. The second responsibility of the screen-manager is to *tokenise* user input.

When the user presses or releases a mouse-button, or moves the mouse with a button down, *while the mouse-cursor is over a graphical object*, an event is generated. (All other events are ignored.) The event is tokenised by generating a token of the form

```
event_type(Window, Button, ObjectID, Position).
```

The following are examples of event tokens. (There is only one window in the current ACORD demonstrator.)

```
button_down(1, middle, 12, 231:125)
```

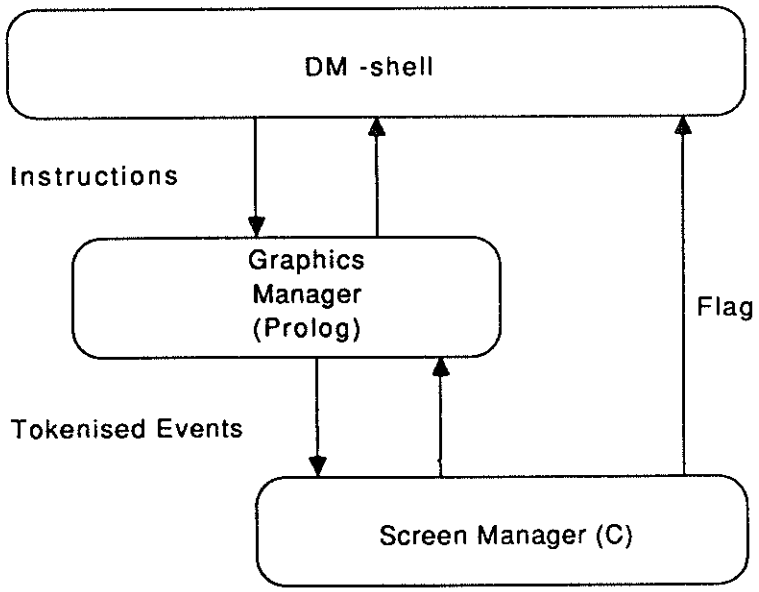


Figure 9.2: Communications in the GS

```
drag(1, middle, 13, 233:126)
button_up(1, middle, 14, 255:132)
```

The tokens are sent to the graphics manager in a sequential string which is submitted to the parser. (A completely separate and differently interpreted kind of token is `chart_report(ChartID)`, which indicates that the user has updated a chart by direct manipulation.) It may well happen that the parsing rules specify feedback to the user — eg. when a truck-icon is dragged and so the cursor must be tracked by a ghost image — in which case calls are made to the screen-manager by the parser. This means that internal GS communications, tokenising, parsing and feedback generation all have to be kept as efficient as possible, so that interaction can happen in real time.

The DM has several other tasks as well as looking after graphical interaction. The other components of the system have to be served also, and hence the DM needs to be informed when a graphical interaction is taking place, so that it can notify the graphics manager of its readiness to accept the resulting GRL expression. There is therefore a 'flag' channel, connecting the screen-manager directly with the GS's DM-shell, which is used to alert the DM whenever an interaction is initiated (see Figure 2).

9.2.4 Further Details of the Components

Implementation of Graphics Manager

Operation The graphics system is designed primarily to be used in the context of the whole ACORD system; however, if the term `standalone` is asserted into the database before the predicate `init/0` is called (in the standard case, before the file `con`, which loads the system, is consulted), the system will function in standalone mode, and calls to `gen_cmd/2` and `g_req/2`, as described below, can be issued at the PROLOG prompt. The predicate `go(N, Result)` invokes a standard example, number N, from the file `examples.pl`, and returns the result, usually as it would be received by the DM.

Rules and parsing As previously noted, the graphics manager handles its input by operating a parser according to a set of "grammar-rules", which are all defined in the file `rules.pl`. These are used in the cases both of input from the user and of input from the DM. We will look at the form of the rules, and then consider these two different uses in turn.

The rule-base The rules used by the GS are essentially a kind of direct-clause grammar (DCG), where the notation is transformed directly into corresponding PROLOG clauses at the time of "consulting" them into the system. An important difference is that the execution of the rules is controlled so as to occur in a breadth-first manner, as discussed below. The transformation is done by the predicate `regenerate_rules`, which also provides a certain amount of consult-time checking of rule syntax. Rules are of the form

```
Head <--> Body
```

where the Body consists of actions, constraints, tokens, and calls to further rules (which can be recursive). Top-level rules appear in `rules.pl` in the special form `rule(Head) <--> Body` to allow the parser to distinguish them. The interpretation of actions and constraints is discussed in the following sections.

An example of a typical rule is

```
move_vehicle(KbId, Obj, Source, Dest) <-->
        grab_object(Obj, Source),
        drag_vehicle(Obj, Loc, Where),
        destination_check(Obj, Loc, Where, Dest).
```

In this particular case, the body consists solely of calls to other rules, which have their own definitions, such as

```
drag_object(Object, EndLoc, EndObjs) <-->
        drag(Window, not_a_button, X:Y, WhatOver),
        action drag_it(Object, X:Y, EndLoc, EndObjs).
```

Here the **drag** term is a token indicating a drag event of which all the parameters except its position (**X:Y**) are ignored. The action **drag_it** invokes a PROLOG predicate which enters a fail-loop controlling the appearance of a ghost image of the Object and its movement to a position returned as EndLoc; in some cases, the icon of an object is animated suggestively while its ghost image is being dragged, so that it is clear which object is involved.

For interest, the definition of **drag_it/4** is as follows.

```
drag_it(Object, X:Y, LX:LY, EndObjs) :-
    obj_id(Id, Object),                          % Find Object's GS Id-num
    runseq(Id),                                  % Animate Object icon?
    ghost_follows_cursor(Object, X:Y), !,        % Create ghost image
    get_next_event([Event|Rest]),                % Get next event
    arg(3, Event, Xd:Yd),                        % Find out its position
    callc(f_move_ghost_to,[Id, Xd, Yd], _),      % Move ghost there
    Event = button_up(_, _, SX:SY, _),           % Backtrack from here
    scaleX(LX, SX), scaleY(LY, SY),              % (From screen co-ords)
    decode_drag_target([Event|Rest], EndObjs),   % Find out where we are
    stopseq(Id),                                 % Stop animation (if run)
    exorcise_ghost, !.                           % Get rid of ghost image
```

It would be preferable for this simple loop to be encoded within the rules, since we could then have more control over what happens, e.g. if the ghost is dragged away from a specified path. At present this is not done because it is too inefficient to permit real-time interaction in all circumstances.

Interpreting input from the user Whenever the user presses a button on the mouse, the screen manager alerts the GS's DM-shell by placing a message on a special "flag" channel. The DM-shell responds by sending the GS a **gop** request. (The term "gop" is an abbreviation of "graphical operations", used for historical reasons.) The GS begins the parsing process by calling **gen_cmd/1** via **parse_user_input/1**.

Parsing of the event stream proceeds asynchronously with the input from the screen manager. Events are queued and made available to the graphics manager through the predicate **get_next_event/1**, which actually returns a list of tokens representing different possible interpretations of a given user-event, e.g. as a hit on either a truck, a city or a road, where depictions of these may all overlap on the screen. Tokens are given in a form which includes only the GS ID-number of an object, along with its screen location, and are decoded into references to named objects by **map_events_to_tokens/2**. The event stream is gradually tapped by the parser failing back into **get_next_event**, which

reads the queue in blocking mode (if there is no event it waits for one) and only stops when the last event received was a button-up (the user let go of the mouse).

The predicate **parse_token/4** takes the token-list and begins a procedure which simulates breadth-first search through the virtual tree defined by the grammar-rules. The first step (implemented by **first_step/2**) is to extract from the rules what *side-effects* are associated with possible tokens in the first level of the search tree. "Side-effects" are *constraints* that have to be satisfied or *actions* that have to be performed in the context of particular rules, and they force the breadth-first search strategy (equivalent to looking at the rules in parallel) because it may well be problematic to undo side-effects that might have been set off by an unsuccessful depth-first search path.

During this step a set of **match_record/5** clauses are **asserted** into the PROLOG database specifying a list of side-effects that would have to be dealt with if a given type of token (e.g. a button-down on a particular truck) were matched in the input, in the context of a particular rule. In effect, for each possible token-match, the state a depth-first search would be in at that point is saved as a **match_record**.

The next step (**next_step/2**) is to find a **match_record** which matches the first *actual* event token. The parse then continues from that point, using the saved parse-state and looking for side-effects of possible further tokens at the next level down, which will be saved in **match_records** as before. When this set of matches is saved, the parser returns to look for other matches with the actual token at the level above. At each sucessfully-matched node in the tree, the set of side-effects is evaluated so that if constraints are satisfied by the given event token the relevant actions are performed. If no matches are found for the current token, others in the same list as returned by **get_next_event** are tried, before the parse will fail.

When the search has been successfully completed, the GRL expression forming the head of the rule that was finally followed through completely is returned to the DM as the first argument to **gen_cmd/2** (the second argument is a **yes/no** flag). Some postprocessing in **parse_user_input** ensures that if the result were a **selection(Object)** term, it is simply queued, since a selection is not treated as a direct-manipulation but may be needed at a later stage by the Resolver in the DM to establish a deictic reference.

Interpreting input from the DM Input from the DM comes in the form of calls to the predicate **g_req/2**, of which the first argument is a GRL expression and the second is used to give a result (usually **ok** or **no**). The GRL expression is handed to **interpret_rule/1**. The graphics manager uses the same breadth-first search through the rule-base as in parsing, this time to find rules whose heads match the GRL expression. Tokens are ignored in this case, and matches proceed essentially on the basis of constraints being satisfied, when any appropriate actions are performed. This is typically a much simpler and faster procedure than parsing user input, since there is little danger of ambiguity or other complications.

Rule support and charts Top-level code for most of the actions available to the rules is to be found in the file `rule_support.pl`; an exception is the chart operations (`charts.pl`). Since an action can be a call to any PROLOG predicate (as, in fact, can a constraint), there is no fixed list of them, but the following are commonly used.

```
create_vehicle/1
get_selections/3
highlight_object/2
locate_moving_vehicle_at/2
locate_along/7
locate_between/4
locate_at/3
```

These actions are more or less specialised to particular situations in which they apply; e.g. `locate_between/4` is used to locate roads between cities, whereas `locate_at/3` can position various kinds of objects in variously specified locations, and `locate_along/7` will place anything at either a specific location or a specific distance- ratio along a route from one city to another. The request to `get_selections/3` will collect a list of N selections from the queue, if there are any: if not, it will prompt the user (in a specified language) to make N selections.

The GS provides charts, usually used to display the loads on trucks or the storage at depots, but in principle of general application. Charts are implemented in such a way that they appear first as icons, which can later be opened (into normal SunView windows) as required. There are several chart-related actions:

```
show_chart/1
open_chart/1
change_chart/4
return_chart_info/2
check_chart_linked/1
check_chart_unlinked/1
```

These respectively create and/or show a chart, open a chart-icon, change the information shown on a chart, return the information shown on a chart, and check whether or not a chart is linked/unlinked to another chart. (The point of the final two is that when a vehicle is placed at a depot, their respective charts are linked so that changes to the vehicle's chart cause corresponding inverse changes to the depot's.)

Implementation of the Screen Manager

This part of the document briefly describes the aspects of the GS software that are implemented in C. It is intended only as a guide, and those who might wish actually to use the system will find no substitute for reading the sources themselves.

Operation The screen manager program is intended only to be run from PROLOG, which can be done either within the whole ACORD system or with the GS in standalone mode as described above. The implementation of its communication channels makes it impossible to run the screen manager as a separate program from the UNIX shell, but there is a standalone test C program which runs it as a sub-process. This is disused, and requires input script files with an obscure syntax. The interested and sufficiently courageous reader is invited to examine `try.c` for further details.

Arrangement of C code There are two kinds of C-functions: those which belong to the screen manager program itself, and those whose object code is loaded into the PROLOG graphics manager, usually for communication purposes. A typical case is that the code for a PROLOG-loaded function simply packages its arguments and writes them down the connecting pipe to a corresponding C-function in the screen manager, thus implementing a kind of remote procedure-call. So typical is this, that an automated system has been produced for generating the PROLOG-loadable "stub" function and a screen-manager "intermediate" function which collects its arguments from the pipe and calls the appropriate "real" function. These stub and intermediate functions are created by a `perl` script called `generate` (in the subdirectory `Scripts`). (`Perl` is a general-purpose manipulation language with abilities like the UNIX standard `awk` but a more sophisticated, C-like syntax.) Definitions of the "real" functions are supplemented with a coded comment of the form

```
Function:name,functype,result,argtypes
```

which is read by `perl` under invocation from the `Makefile`. Stub functions in `$SUN_ACORD` go into `stubs.c` and intermediates into `intermediates.c`, and `stubs.o` is eventually loaded into PROLOG. By convention, all stub functions have `f_` prepended to their "real" name; they are identified in the message sent down the pipe by code-numbers also created by `generate`, in the file `$SUN_ACORD/h/functions.h`. Lists of the PROLOG-callable functions — the "remote" `f_*` functions and the other "local" functions used e.g. for spawning the screen-manager process — are saved into `$SUN_ACORD/pl_loadfile`, which is consulted by the graphics manager at system startup. Any of the stub functions can be accessed individually from PROLOG using the built-in `callc` predicate, bearing in mind that arguments must be carefully checked for number and type.

Data structuring in the Screen Manager The screen manager is implemented along roughly object-oriented lines. A number of types of object are distinguished, each of which has a corresponding C structure and a set of functions mostly kept "private" under static storage related to particular source files. Principal object types are *windows, icons, lines, nodes, relations, texts* and of course *charts*, each of which has its own readily-recognisable `.c` file (and sometimes some other related ones). Obviously, each object type has some "public" functions which provide its interface for the rest of the system; their names are in general constructed conventionally as `<function>_<obj-type>`. Usually, there is a `create_` function, a `highlight_` function which causes the object's

screen representation to flash on and off, a set of drawing-related functions such as `draw_`, `delete_`, perhaps `move_`, `change_`, etc. Created objects are kept as linked-lists of structures in a standard way.

Relations A fundamental notion in the screen manager, included above as an object-type, is that of a *relation*. Relations are set up to constrain or define the positions of objects on the screen. The set of relations available is given in the enumeration `Relationship`, in `h/relations.h`. Currently, the set consists of RELATIVE, BETWEEN, ALONG, IN, AT and NEAR. AT is a basic relationship between an object and a location (screen co-ordinate); the rest relate two or more objects to each other. In the case of RELATIVE the relationship is explicitly parameterised by a position-vector, and ALONG by a distance-ratio. The others have automatically derived positioning; the commonly-used NEAR attempts to place the related object (e.g. a text as the name of a city) beside the target object (the city) but without overlapping any other objects in the vicinity (except lines). (In the worst case, this defaults to placing the related object centrally on top of the target.)

Whenever an object has its position updated, the relations it stands in have to be examined to check that they still hold. If necessary, the object or some others will have to be moved. In general, the structure of relations defines a sort of "tree", and an update to an object means that all objects in the tree below it should be checked and possibly repositioned.

Calls to `f_create_<object-type>` and then `f_create_relation` are the main way that the PROLOG part of the system builds up a screen image and its corresponding screen manager structures. Objects (e.g. icons representing trucks) can be moved simply by changing their relations. There are of course also a number of special-purpose functions such as `f_set_image_sequence` which sets an icon to cycle through a given sequence of animation icons (if present in the subdirectory `Icons`) to produce the effect of a moving image.

Hooks When objects have to be repositioned there are certain functions that obviously need to be generally available, e.g. to find the sizes of other objects which have to be checked for occlusion. Similarly, highlighting involves a periodic search (set off by a timer event from the SunView system, actually several times per second) to see which objects have to be flashed. These generalised functions are made available by using tables (arrays) containing functions of a particular kind for each object type. These functions are conventionally named as

`<object_type>_<function>_hook`

where the object type is abbreviated to give e.g. `nd_size_hook`, `im_highlight_hook`, etc. The tables are automatically constructed by the same `perl` script that builds the stub functions, using a similar syntax for comments embedded in the hook function's

source file. Files named as `tab_<function>_hook.c` are created in the `$SUN_ACORD` directory.

Input events Input events are handled, in the usual SunView manner, by an event procedure triggered by an interrupt whenever an event is received. Essentially, events are simply queued for the graphics manager, but first they are "tokenised" by searching the space of known objects displayed on the screen to find any within whose boundaries the point at which the event occurred lies. The GS ID-number of such an object is used in the event token, as described in previous sections, and if there is more than one object then more than one token is generated and the list of them is queued. At the same time, if the event is a button down, an alerting message is put on the "flag" channel for the DM-shell.

Events for charts are handled differently, however, being neither tokenised nor queued but treated by event procedures special to the subwindows in which charts occur. Only when the user has pressed the "exit" button on a chart is a chart-report event queued and a message set on the flag channel.

9.3 Relations to other ACORD components

In this section we describe the nature of the I/O at the interface between the GS and the DM. This consists primarily of the Graphical Representation Language (GRL), which is based on the DRS version of InL used in the DM. The complete set of GRL expressions recognised is in fact specified by the top-level parsing rules in `rules.pl` and hence is very malleable and extensible.

9.3.1 Objects

Objects are described just as in DRS, using PROLOG terms of the form `obj(KbId, Sort, Class, Name)`. The GS stores all the information contained here as the third argument of a `obj_kinfo/3` clause, one of which is asserted into the GS database for each object. In general, nothing is done with the *Sort* information, except that it may be used to detect plural objects. Whenever information about an object is given back to the DM, the complete `obj/4` term is reconstituted.

9.3.2 Actions

GRL is thought of as specifying actions that involve one or more objects as arguments. The general form is therefore `action(Obj1, Obj2, ...)`. However, in some rules arguments may also include PROLOG lists of objects, numbers, etc.

Maps and charts The only action that applies to maps is **draw**; charts also have **show** and **change**. The latter may be sent either from the DM as a command, or to the DM as a result of direct manipulation by the user. The ID given by the DM to a map or chart is in fact an **obj/4** term.

Vehicles The principal action for vehicles is **move**, with four arguments:

```
move(BackgroundMap, Vehicle, Source, Destination).
```

A vehicle will be created if it is unknown when first referred to by an instruction from the DM, and visualised depending on its *class*, given as the third argument of its **obj/4** term, which currently might be either **truck** or **tanker** (**vehicle** defaulting to "truck"). The main function of the **change** action as applied to vehicles is to change the class or name, e.g. when new information comes to light identifying something previously known only to be a "vehicle" as a "tanker"; it currently may never originate from the user, only the DM. Either the Source or the Destination of **move** may be unspecified (i.e. specified as the empty list '**[ ]**'), but not both: the source has no graphical effect whether specified or not, whereas an unspecified destination leads to no movement but instead the animation of the truck icon at its source to indicate that its position is uncertain.

9.3.3 Selections

An important feature of the GS is the provision of some means for the DM to get information about objects selected by the user, e.g. as referents for deictic NL expressions. This is allowed for by the request for **hits**, which returns a list of selections as its third argument. For technical reasons in the DM it is convenient to encode these selections in a special syntax, which takes the form

```
Vehicles::Cities::Roads::Maps
```

where the components (any of which may be the empty list) indicate items lying under the cursor at the point of the hit. However, this is a fairly unattractive syntax because of its relative commitment to the current application structure, and should ideally be replaced with something more general (such as a simple list of objects).

9.3.4 Miscellaneous

Other rules are intended to give the DM some control over various features of the operation of the GS. Highlighting of objects on the screen is available through **highlight**

(ListOfObjects) and selections can be flushed to reduce the risk of corruption from parts of earlier interactions.

The relationship between the GS and KB should be noted, especially with respect to the handling of charts. Information on charts comes in and out of the GS essentially in the form of arrays. Whenever an update is performed to any part of a chart, the array for the whole chart is sent to the KB. The KB maintains its own record of what is displayed on a chart (although that chart may not be visible at any particular time), and also of any links (i.e. between trucks and depots) that affect updates to it possibly resulting from the interpretation of NL expressions. Thus, although the data structures are quite independent, the system functions as though the KB were using the chart information actually stored in the GS. This arrangement somewhat bypasses the visualisation component in the DM and is for that reason not altogether ideal.

9.4 Potential of Development

The ACORD Graphics System is a simple piece of software. We make no bold claims about its reusability or the appropriateness of trying to make it work in any other context than the whole ACORD demonstrator system. Its interest lies primarily in the approach to the treatment of graphical dialogue embodied in its design. We believe that this is an approach with a future, and that software based on similar principles will eventually enter general use; but considerable further research is needed before this can happen. It is important to note that this is what one would expect and what ought to be the case. A project like ACORD, with large research ambitions at the edge of our understanding of what can be formalised and programmed at all, would be misconceived if it were burdened with the futile goal of producing serious software within its own timespan.

Especially where there is any real prospect of the software being of general use, we have tried to make it as flexible and extensible as possible. This is true particularly of the parts written in C, which form a very general type of screen manager for C-Prolog programs, its usability being considerably enhanced by the automatic generation of "stub" functions etc. It effectively provides a "toolbox" of functions that can be easily customised to other requirements. Although some of the functionality might be seen as superseded by developments in server-based window managers, e.g. X-Windows, we expect to use some of these parts of the system in other applications.

The PROLOG graphics manager is really only to be seen as an experimental system for looking at parser-based user interfaces. As such, it has the same merits of adaptability, a very convenient rule syntax, etc. However, all such software necessarily has a very short working life: it is too simple to be of much use for very long without extensive rewriting; even before it is finished, the ideas behind it have progressed beyond the momentary "snapshot" view of research that it encapsulates. This, of course, is precisely why it is developed in a malleable language like PROLOG.

Chapter 10

Theorem Prover

Fariba Ommani
BULL

This chapter presents the Theorem Prover founded in the tableau method. it uses unification to find the adequate substitution of the universally quantified variables, while preventing infinite loops by using heuristics.

10.1 Scientific Background and Development

Introduction Two main directions have been followed during our contribution to the ACORD project:

The first is the exhaustive work done in the theorem proving domain which has resulted in a standard full first order theorem prover called **PTP**(see Ommani and Sedogbo 88).

The second is the work done on integrating the **PTP** to the ACORD prototype which resulted in the **ATP** (ACORD's Theorem Prover).

The PTP system This part of our task concerns the studies and efforts accomplished to obtain an efficient theorem prover based on tableau calculus. The tableau method has two inconveniences when implemented as described by Beth 59 and Smullyan 68. The first inconvenience is the way they eliminate universal quantification which implies multiple substitution of universally quantified formulas with all elements of Herbrand's universe until the appropriate one which will produce a contradiction in the tableau is found. The second inconvenience concerns the generation of infinite trees for some non-theorems.

In **PTP** a universal variable is substituted only once and with the PROLOG term **tt(X,Ub)** where **X** is a PROLOG variable and **Ub** is the upper bound of Herbrand's universe. The special unification rules of **PTP** then bind the PROLOG variable **X** with the proper Herbrand term.

The second advantage of **PTP** is that it prevents the infinite tree construction. The infinite tree is produced for a non-theorem whenever each recapitulation of its universally quantified formulas introduces some new terms in Herbrand's universe (Boolos 84). An informal discussion presented in Deliverable T1.8b shows that in **PTP** there is no need to do the recapitulation more than once in each branch of the tableau. Another interesting point developed during the work on theorem proving is the breadth-first strategy.

In this strategy a real tableau is built for each formula. Thus in the case that a formula is common to several branches, it will be treated only once and propagated to other branches. This makes **PTP** very efficient especially for the domains where this case occurs frequently. Its drawback regarding the breadth-first strategy is that it needs more run-time memory space.

Breath-first strategy and depth-first strategy can be adopted depending on the applications. For example the special structure of the formulas in the ACORD system which usually invokes the common formulas in the different branches make the breadth-first strategy more efficient than the other one.

The ATP system The **ATP** system is an adaptation of the **PTP** system to the ACORD prototype. However for efficiency reasons the universal variables are substituted with PROLOG variables. This simplification of **PTP** is permitted because in the ACORD system the universe is supposed to be finite (Deliverable T1.8b).

The input of **ATP** is made up of formulas expressed in a special semantic representation language called InL. They are translated by **ATP**'s translator to the first order predicate logic format.

There are several special aspects developed in **ATP** to integrate it to the ACORD prototype: an indexation mechanism, KB interaction, plurals and consistency maintenance.

Indexation mechanism The choice of the relevant premises to be decomposed in the proof procedure is important when we have to deal with an arbitrary set **K** of formulas and this is the case of the ACORD system. The idea is to index each element **F** of **K** by literal formulas that are produced by constructing the tableau for **F** (Deliverable T1.2b). This indexation is initially done for all formulas of **K**, but is repeated whenever **K** changes. The tableau for **F** gives the opportunity to index **F** by the polarity (or sign) of literal formulas of branches. So each literal is linked to all formulas which produce it by the PROLOG predicate *index(Literal, List_of_Formulae)*. Whenever a statement is to be proved, the list of formulas to be put in the premise of the proof procedure is obtained by calling the *index(not Literal, List_of_Formulae)* where **Literal** is a literal

obtained by decomposing the statement.

Knowledge Base interaction As described in the general architecture, the sentences updated in the ACORD system are stored in two different Bases. Simple facts are stored in the Factual Base of the KB component and complex sentences are stored in the **ATP** Local Base. In addition the KB stores the World Knowledge necessary for conceptual reasoning on the texts (see below **11**).

Therefore the **ATP** system should find the premises of a proof in two distinct bases. The first one is a KB having its corresponding representation language and inference mechanism. The second one is **ATP**'s Local Base which stores the complex updates in a first order predicate language. Through the interface predicates all the knowledge stored in the KB can be entered in the **ATP** proof procedure. From the **ATP** point of view the KB can be considered as a PROLOG database. So, whenever an atomic fact **P** is obtained in a tableau, the **ATP** calls the interfacing predicate *ta_verify (P, SAT_list)*, which searches for the fact **P** and returns the list of results in *SAT_list*.

Consistency maintenance The **ATP** is responsible for verifying whether a new update is consistent with respect to what has been already entered in the system. For each new update in the two bases of the system (the KB or the **ATP**'s Local-Base), the **ATP** tries to verify if there is a contradiction between the update and what is already entered in the system. In the case of a contradiction the statements which produce the inconsistency are eliminated. As the updates in the system are stored in two different bases the consistency check is done by the **ATP** in tight interaction with the KB. Two different kinds of update can be considered:

(1) Positive and negative statements: Before the assertion of a sentence in the KB the DM asks the **ATP** if the sentence is not in contradiction with the current state of the system. To do this the **ATP** tries to verify the negation of the update. If it is proved, it means that the insertion of the update to the system will produce an inconsistency in the system. A trace of the proof is kept during the proof procedure to find those statements already entered in the system which are responsible for the contradiction. In the case of contradiction these facts are dropped from the KB and the **ATP**'s Local-Base, before the insertion of the sentence.

It must be noted that the consistency maintenance in a natural language system is not limited only to the detection of the logical contradiction. Conceptual contradiction is another aspect to be considered in the consistency check procedure. In the ACORD system the conceptual contradictions are defined in the KB by means of some conceptual consistency rules (Cf. TA's contribution). These rules are applied whenever the KB is asked by the **ATP** to verify a fact (see below **11**).

(2) Rule statements: We call rule statements sentences like *All trucks which transport*

printers go to Berlin. Rule statements are considered in our system as assertions (which are the consequences of rules) about a set of objects defined by the premises of the rules, because we consider that a general rule statement is conceived in the conceptual part of the KB at the beginning and is fixed during the user sessions. For instance in the example above *all trucks* refers to a finite set of already known trucks and does not express a general rule about trucks.

On the other hand to be able to maintain the consistency it is necessary to apply the rule statements when they are entered in order to be able to check their consequences on the current state of the system. Moreover this is the only way to make the system capable of carrying out the graphical update of a rule statement. So if we represent the rule statement as *if P then P implies Q.* the sentence above is understood by the system as *Q of a set of trucks* verifying the condition P.

The way the **ATP** proceeds to do this is the following. First it tries to prove **P** and finds all truck instances which verify **P**. In the second step it instantiates **Q** for the set of trucks and makes an update of the new statement - which will be a singular in the case where one truck instance is found - to the system.

Plurals Generally the transformation of plural sentences in the logical formalism produces the predication on a set of individuals. This goes beyond the classical techniques of theorem proving which accept only the predication on individuals. In the ACORD system only a distributive reading of plurals is accepted. This means that we consider that a plural sentence is a conjunction of several separate sentences. A way to deal with this is to use rules - called specification and generalization rules - which translate higher order predicates to first order predicates. However the application of these rules is inefficient and not always necessary.

In the **ATP** system the application of the rules is avoided when it is unnecessary. For example while a coordination is split into separate formulas, a sentence like *7 trucks are in Paris* is stored as a whole. The point is that in our special plural representation (Ommani and Van Hoof 89) we have avoided the predication on plural Reference Markers. A plural sentence is represented as *set(S,E,INL1,INL2,C)* where **S** is the set of all elements **E** that fulfill *INL1* and *INL2* and **C** is the cardinality of **S**. While **S** is a plural REFM, **E** is a special singular REFM called *abstract variable* which is used to refer to the plural objects defined by **S**. The plural Reference Markers do not appear in *INL1* and *INL2*. The plural noun-phrases are represented inside it INL1 and *INL2* only by their corresponding *abstract variable.*

From the **ATP** proof procedure point of view an *abstract variable* is not different from other singular variables. They can be unified by other singular or abstracted variables through the proof procedure. The difference is that each *abstract variable* corresponding to a plural noun-phrase is linked to its plural REFM. The update of the plural sentence is different for the other sentences. Whenever a set structure is entered in the system it is stored in different parts of the system. InLs are stored in the KB when the information on the set is stored in a table called *set_table.* In the same way when a plural question

is entered the **ATP** proceeds in two steps. First it tries to prove the InLs and finds the corresponding answers. The second step consists of finding the information concerning the sets for the *abstract variables* which are found in the list of the answers.

The special treatment of the answers directly obtained by the **ATP** allows it to deal efficiently with *howmany* and *howmuch* questions. It also makes the system capable to behave as a semi-modal system concerning *yes/no* questions dealing with quantities. In the following paragraph we will show through a brief example how this works exactly.

10.2 Technical Description

10.2.1 Software and Hardware Requirements

Both versions of the Theorem Prover (the standard one and the ACORD one) are implemented in C-Prolog and are relatively small in terms of disk space. But their run-time memory allowance is relatively high because of the breadth first strategy used.
Following are the technical details concerning the memory allocation:

Disk storage & 40 kbytes for **PTP**
Disk storage & 150 kbytes for **ATP**
Run-time heap & 1500 kbytes

10.2.2 Files

Code concerning **PTP** can be found in the *bull/t_prover* directory. The file *envir* contains the user interface program and *tp_prover* contains the main programme. *theorems* is the file which contains the list of theorems proved by **PTP**.

Code concerning **ATP** is in the *bull* directory. The list of files to be loaded is in the load file *sys_tp*. The file *tp_manager* contains the principal calls concerning query answering. It calls the predicates of the files *tp_traduct* and *tp_preps* to prepare the question, *p_prover* to prove the formula and *tp_respond* to prepare the answer.

The file *tp_consist* contains the program which tests and maintains the consistency of the system. The file *tp_kb_inter* contains the program which does extended unification between the objects of different classes but the same super-class.

10.2.3 Software Description: General Design

Implementation strategy The **PTP** has been tested on various well-known logic formulas; all the theorems listed by Pelletier 86 and Kalish and Montague 64, except those with identity and functions, have been proved.The **PTP** accepts a singular formula and calculates its truth value. But it is not efficient when it is used to prove a statement as a consequence of a large base of formulas. This is because the **PTP** does not actually benefit from an indexation mechanism to find the relevant formulas in a base. This mechanism is used for the **ATP** and is capable of dealing with a large base of formulas.

User interfaces Running the **PTP** is easy. The file to be loaded is 'load_tp'. By calling the predicate 'go' the user is invited to enter his or her formula in the standard first order format.

10.2.4 Illustrative Example

In this paragraph, we show through an example how the **ATP** intervenes and how it proceeds during the session given the following interactions:

USER: 5 trucks transport 500 printers.

The semantic representation of the sentence can be illustrated as the following:

```
set(SET_RefM1,ABS_VAR1,[truck(ABS_VAR1)],
     [transport(ABS_VAR1,ABS_VAR2),
      set(SET_RefM2,ABS_VAR2,[printer(ABS_VAR2)],[],500),
     5)
```

For simplicity reasons we have not exactly respected the InL representation. To update this sentence the **ATP** is called by the DM to check the consistency of the system. The **ATP** tries to find the negation of the sentence. In this case there is no inconsistency between the sentence and the current state of the system.

The parts of the sentence which are stored in the KB are the facts:

truck(ABS_VAR1)
transport(ABS_VAR1,ABS_VAR2)
printer(ABS_VAR2)

The pieces concerning the set information are:

set_table (SET_RefM1, ABS_VAR1, 5, [])
set_table (SET_RefM2, ABS_VAR2, 500, [SET_RefM1])

Also note that the *abstracted variables* are in fact of the form:

ABS_VAR1 = term (100, sing, ..., printer, SET_RefM1)
ABS_Var2 = term (120, sing, ..., truck, SET_RefM2)

The system informs the user of the end of the update by 'OK'.

Now consider the second interaction as:

How many printers do the trucks transport?

First the RESOLVER calls the **ATP** to find the reference of *the trucks.* It asks if the trucks which transport printers exist The **ATP** finds the answer **ABS_VAR1** for the RESOLVERThe resolved sentence is then passed through the DM to the **ATP**. This is the semantic representation of the question:

```
set(SET_RefM,ABS_VAR1,[trucks(ABS_VAR1)],
    [transport(ABS_VAR1,ABS_VAR*),
     set(SET_RefM*,ABS_VAR*,[printer(ABS_VAR*)],[],X)
    5)
```

First the **ATP** tries to find the answer for the conjunction of the facts:

truck(ABS_VAR1)
transport(ABS_VAR1,ABS_VAR)*
printer(ABS_VAR)*

The above formula is proved by the **ATP** and the answer is *ABS_VAR2.* The second step then consists of consulting the *set-table* to evaluate the cardinality of the printers. The *set_table* shows that the cardinality of the printers is 500. On the other hand it refers to *SET_RefM2.* This indicates that the set of printers is quantified by the set of trucks. So the cardinality of the set of trucks should also be taken into account when the cardinality of the printers is to be found. This is computed by multiplying the cardinality of the printers with that of the trucks(i.e. 2500).

10.3 Relations to Other ACORD Components

Because of its hybrid architecture the **ATP** is highly dependent on other modules of the ACORD system. It has two interface predicates with the DM. They are the input/output predicates of the system. *bull_update_drs(INL, Consequence_INL)* is called by the DM to update a complex sentence. **INL** is the semantic representation corresponding to the sentence to be updated and *Consequence_INL* is an InL representing the consequence of the sentence in the cases where the sentence to be updated has an implication meaning. *qu_tp_prove(Question, Result)* is the second predicate interfacing the **ATP** to the DM where *Question* is the query and *Result* is the extracted answer.

The other module cooperating with the **ATP** is the Knowledge Base. The main predicate interfacing these modules is *ta_verify* which is called by the **ATP**. This predicate is used to integrate the conceptual reasoning mechanism of the KB into the **ATP**. *tp_verify_drs* is a predicate called by the KB to verify the consistency of the whole system when it has to update a new fact. *ta_comp_obj* and *ta_comp_class* are two other predicates called by the **ATP**. They are used to extend the unification mechanism of the **ATP** to the objects having the same class.

10.4 Potential of Development

Use in other situations The **PTP** can be used as an independent standard Theorem Prover but the **ATP** can only be used with the DM and the KB because of its tight dependency with its internal language and the KB.

Chapter 11

Knowledge Base

Peter Ovenhausen
TA

The Knowledge Base, storing natural language, graphics information and conceptual knowledge, is described in this chapter.

11.1 Scientific Background and Development

Most of the larger natural language systems of the past ten years focus on querying a knowledge or data base using more or less involved techniques of semantic representation and dialogue management. Some of those systems are Siemen's CONDOR, the University of Hamburg's HAM-RPM, IBM's USL, and SRI's TEAM and Core Language Engine. ACORD was one of the very first projects that explicitly addressed the construction and interrogation of a knowledge base. In addition, ACORD was also one of the first projects that investigated the integrated use of natural language text and graphics. Both problems have been placed within a possible use of French, German, and English as natural languages. Hence, ACORD can be said to aim at a multi–functional, multi–modal, and multi–lingual interface.

The domain of business communications was the first area of application that has been selected for the reason that business communications, typically business news in a newspaper, might best satisfy the three main features of ACORD: considering business news of French, German, and English newspapers, a mono–lingual user of one of the three languages might well be interested in updating his knowledge base of the European economy (possibly using electronic file transfer, or a scanner), and then interrogating it in his native language on the facts and economic developments he is interested in. As a number of economic facts and developments can conveniently be presented in a

pictorial fashion, the use of graphics and deictic interaction is a natural extension of the user–interface for this kind of application.

In order to restrict the domain of application, it was agreed to only consider business communications relating to the stock market, i.e. company news, stock rates, and financial company reports; see Deliverable T5.1 for a detailed description. As it turned out, however, this kind of application was both too demanding and too simple. From the point of view of syntactic processing, reading in free newspaper business communications in French, German, and English (that typically include a large number of possibly unknown proper names of persons, companies, and products) appeared to be too difficult to process. On the other hand, most of the business news under consideration could be traced to simple numerical relationships so that the domain was felt to lack an interesting semantics as was indicated by the fact that most of the graphical depictions were reduced to simple bar or pie charts.

In a second application, the ACORD interface was placed in the context of a decision support system concerning shipments of goods by trucks in Europe. It was assumed that a multi–national carrier keeps a central knowledge base of all facts, rules, and regulations concerning its trucks, shipments, and orders. (Considering the still numerous restrictions and particularities on commercial traffic in the individual European countries, such a knowledge base might in fact be of practical use). New facts, say on the position or cargo of a truck, can then be added in any one of the three ACORD languages or by using graphical means. Conversely, this knowledge base — assumed always to be in the most up-to-date state — can be interrogated in any one of the three ACORD languages or by using graphic interaction, irrespectively of the mode and the language by which new knowledge has been added. Since this application does not involve the processing of high-level free text, it was deemed syntactically managable, while the domain of application is semantically complex enough to provide for interesting graphical and deictical applications. Deliverable T5.2 treats in detail the kinds of charts encountered in the new area of application, and discusses possibilities of processing them.

From the point of view of the knowledge base, this application has lead to a number of problems of which the most important are:

1. Delimiting the range and content of the intended application
2. Defining the relation between semantic representation and knowledge base
3. Defining a formalism for the representation of knowledge that is independent from the language of update and query
4. Defining a formalism for the representation of knowledge that is independent from the mode of update and query
5. Defining a formalism for the representation of conceptual knowledge

(1) Given that the ACORD system was intended as a multi–functional, multi–modal, and multi–lingual interface that could be used in connection with data base applications, complex expert systems, or decision support systems, a choice had to be made between

the effort that was to be put in the actual application of the demonstrator system, and the efforts for defining and implementing a knowledge representation that fulfills the intended functionality of ACORD as an advanced multi–purpose interface. It was decided to take a radical solution, restricting ACORD strictly to an investigation of advanced interfaces, and not to build an expert system, or decision support system, for the European transport business, with the exception of a minimal modelling of the domain of application as it is necessary for demonstrator purposes.

The fact that such a choice had to be made indicates that the design of a general interface is not feasible; instead, interface and application should always be developed together. Nevertheless, it is to be hoped that experiences and results from ACORD will be integrated into advanced standard user–interface toolkits like ANDREW, XTool, or Motif.

(2) Given a reasonable complexity of the semantics of a domain of application, most natural language processing systems translate the natural language input into a semantic representation which then is further processed. In the case of a query–system, the semantic representation is usually passed on to a knowledge or data base manager that will try to retrieve the requested information from the data or knowledge base. The distinction between a semantic representation of input and information kept in a data or knowledge base is particularly useful when the natural language queries may lead to complex data or knowledge base system calls as is the case with queries involving (multiple) quantification. In the case of a system where natural language (or graphics) may also be used for updating the data or knowledge base, the situation is different, however. For it amounts to a superfluous duplication of data when for every input its semantic representation is also translated into the knowledge base. Hence, it needs to be clearly defined which representation formalism will be used for the representation of updates, and how the different representation formalisms, if any, will interact to retrieve the requested information.

The delimitation of semantics and knowledge representation has been one of the most intricate problems in ACORD. The solution proposed and implemented for the purposes of ACORD is based on a distinction between four kinds of knowledge, resulting from the four possible combinations of the parameters factual vs. conceptual, and object vs. relation:

	factual	**conceptual**
relation	InL–term	relation–frame
object	instance	object–frame

The relation between semantics and knowledge representation is explicitly discussed in Deliverable T5.4; the actual implementations are described in Deliverables T5.6, T5.7, T5.8, and T5.10. The related problem of how to retain consistency with respect to updates is treated in Deliverable T5.8.

(3) In order to fulfill the task of multi–linguality, the concepts and relations of the knowledge base need to be represented in a terminology that can serve as the basis for a translation into the individual languages. To this end a lexicon of knowledge base terms

specified in "knowledge–base–English" for the domain of application has been defined that also serves as interlingua between the three ACORD languages for the linguistic modules; see Deliverable T5.5 for details. This approach no doubt has its limitations, but appears to be suitable for the purposes of the demonstrator system. In practice, the definition of a common lexicon for the three languages and the knowledge base has proven to be a useful tool for checking the functionality of individual modules.

(4) One of the major aims of ACORD is to support advanced forms of system–user interaction using natural language text and graphics. The following kinds of interactions are possible: Pure natural language interactions, pure chart interactions, i.e. the user points to items on the screen or changes them by using the ACORD graphics system, and mixed natural language and chart information. Hence, the knowledge base needs to provide an abstract representation of information that is independent from the mode that information is presented to, or by, the system. Notice that this problem cannot be solved by simply including in the knowledge base (digitalized) pictorial data, as is offered by some knowledge representation systems. Rather, the problem is to define a representation by which natural language expressions can be assigned the same semantics as certain graphical depictions.

The proposal followed by ACORD, outlined in Deliverable T5.8, proceeds on the basis of a distinction between the meaning and the means of charts. The meaning of charts, i.e. the information which is depicted by a chart, is stored in a Fact Base containing drs–terms as described in Deliverable T5.7. Thus, the meaning of charts is stored in the same way as information which is delivered by a natural language statement. Information concerning the chart itself, i.e. the means of charts, is represented using a chart table that contains an entry for each chart that is currently accessible on the screen. Each time a new chart is depicted, a new entry in the chart table is added, where those entries consist of different pieces of information, including the chart identifier, the term address list, the chart description, and the X-label and the Y-label.

(5) An advanced question–answer system should provide facilities for answering not only those kinds of requests where the query directly matches some stored information, but also those more complicated, but also more realistic, cases where some "thinking" is required, i.e. where the requested information needs to be inferred on the basis of already known, and available, information. Most work on the knowledge base has been spent on defining and implementing a formalism for representing and processing conceptual knowledge. A first proposal, presented in Deliverable T5.3, has been substantially revised. The conceptual knowledge base of the demonstrator system follows the KRYPTON knowledge representation paradigm, and basically consists of an advanced frame system, and a number of rules used for the resolution of entailments and presuppositions; for details see our scientific contribution in this volume, and Deliverables T5.4, and T5.5. The conceptual knowledge base can be considered to provide a lexical semantics for the individual and predicate constants of the semantical representation (InL).

11.2 Technical Description

11.2.1 Software and Hardware Requirements

The KB component of ACORD is implemented in PROLOG, and runs under various dialects of that language. The storage requirement of the component is as follows (these figures are approximate and are given for prolog+):

Disk storage for program	141	Kbytes
Disk storage for frames	56	Kbytes
Run-time heap	900	Kbytes
Run-time atom space	350	Kbytes

11.2.2 Files

There are 31 files which include the source code of the knowledge base component. These files are stored in the directory `~acord/acord/ta`. In the following we describe the files and comment on the task that they contribute.

Load files

`sys_kb`	contains the consulting calls for all KB-files
`load_cr_up`	contains the consulting calls for the files which deal with update and query
`load_alone`	contains the consulting calls for all files which are necessary for the stand-alone version of the KB
`load_tp`	contains the consulting calls for all files of the TPwhich are necessary for the stand-alone version of the KB

Direct access to stored knowledge

`utilities`	contains the code for the commands to list and delete factual knowledge
`show`	contains the code for the command to list conceptual knowledge

Update of factual knowledge

`up`	contains the main code for updating factual knowledge
`conjoin`	contains the code which splits updates containing *and*
`valid_opposite`	contains the code for consistency checks concerning non-numerical information
`consis_number`	contains the code for consistency checks concerning numerical information
`obj_term_table`	contains the code which handles a table relating objects and terms in which they occur

Query of factual knowledge

`cr_main`	contains the code which distributes queries to the appropriate inferences
`cr_rel`	contains the code which performs conceptual reasoning to answer relation–terms occuring in a query
`cr_adj_one`	contains the code which performs conceptual reasoning to answer adjunct–terms occuring in a query
`cr_comp`	contains the code which performs conceptual reasoning to answer component–terms occuring in a query
`cr_measure`	contains the code which performs conceptual reasoning to answer measure–terms occuring in a query
`cr_inst`	contains the code which performs conceptual reasoning to answer instance–terms occuring in a query
`ccr`	contains conceptual consistency rules
`tools`	contains predicates which are often needed by the conceptual reasoning

Chart treatment

`chart_treat`	contains the code which deals with the update and query of chart–information
`chart_table`	contains the code which handles a table concerning the charts

Access to conceptual knowledge

`pro_stat`	contains the routines which give access to conceptual knowledge
`anschl_fr`	contains the routines which reads and stores conceptual knowledge which is stored in a special format

Conceptual knowledge

`kb_ck_stat`	is the knowledge base of the conceptual knowledge which is stored in a static format, i.e. whithout dynamic calculation of conceptual values
`kb_ck_stat.ADD`	is an addition to the knowledge base of the conceptual knowledge

Services for other modules

`cr_task`	contains some conceptual inferences which ask queries of the RESOLVER concerning the reference of expressions
`resolve_interface`	contains some plausibility checks which are done to check the results of the RESOLVER
`visu_info`	contains the code which extracts and formats the information delivered to the visualization component
`tp_ck_interface`	contains some conceptual inferences which ask queries of the TP concerning the matching of objects
`class_suprem`	contains some routines which help to answer the queries of the TP
`def_descr`	contains some routines which collect information about an object in order to help the GENERATOR to generate a definite description

11.2.3 The Stand-Alone Version versus the Integrated Version

The KB can run as an integrated module of the ACORD system or it can be a stand–alone system.

If the KB is to work as a component of the running ACORD system a special process is created which starts PROLOG and consults the file **`sys_kb`** which loads all files of the KB.

Especially for testing, debugging, and improving the KB it is more convenient to have a stand–alone version. This version will be available by starting /prolog and then typing `[load_alone].` which consults all files of the KB and the necessary files of the TP. The TP is necessary because the consistency check which will be done for each update activates the TP. After loading the stand-alone version of the KB each predicate can be called and activated by typing. Furthermore special factual and/or conceptual knowledge can be directly added or removed. Thus the stand–alone version simplifies debugging and implementing new features.

11.2.4 The Different Knowledge Types and Their Representation

The KB deals with three different kinds of knowledge: *Factual Knowledge*, *Chart Knowledge*, and *Conceptual Knowledge*. In the following each kind of knowledge will be discussed by showing what is meant by the term and by explaining the representation.

Factual knowledge The Factual Knowledge is the knowledge which is introduced by the user by typing natural language assertions or by performing graphical actions which are interpreted as new assertions. In the following such a new input to the system given by the user will be called *update*.

The Factual Knowledge is represented by stored terms and tables. It is worth to mention that the terms correspond directly to the drs–terms which are based on the InL-terms. Now the different terms are described.

Name:	instance term or inst–term
Predicate:	`ta_kb_inst`(<Ident-Nr>,<Sort>,<Class>, <Ref–Marker>)
Example:	`ta_kb_inst(104,[1,_,1,0],human,john)`
Explanation:	An inst–term is stored for each item that occurs in a sentence and also for an event. The set of all inst–terms provides the set of all instances which are explicitly mentioned in updates and can serve as antecedents for refering expressions.

Name:	relation term or rel–term
Predicate:	`ta_kb_af_rel`(<Rel-Name>, <Event-Obj>, <Subject-Obj>, <Dir-Object-Obj>, <Indir-Object-Obj>)

Example:	`ta_kb_af_rel(<transport,`
	`obj(100,[0,_,1,0],transport,_),`
	`obj(101,[1,_,1,0],truck,truck1),`
	`obj(102,[1,_,1,0],computer,_),`
	`0)`

Explanation:	The rel–term desribes an event by relating the participating instances to the syntactic functions they fulfil. Furthermore a rel-term gives the sentence predicate and the corresponding event object. That means the structure of a rel-term corresponds to the structure of a simple main clause.

Name:	adjunct term or adj–term
Predicate:	`ta_kb_af_adj( [<Adj-Name>],<First-Obj>,<Second-Obj>)`

Example	`ta_kb_af_adj([spat(loc(in))],`
	`obj(101,[1,_,1,0],truck,truck1),`
	`obj(103,[1,_,1,0],city,paris) )`

Explanation:	The adj–term describes adverbial relations between instances and between events and instances. The adverbial relations cover locational relations, time relations, goal and source relations.

Name:	measure term or measure–term
Predicate:	`ta_kb_af_measure(<` Substance-Obj>,<Unit>,<Number>)
Example:	`ta_kb_af_measure(obj(107,[1,1,1,0],wine,_),liter,15)`
Explanation:	The measure–term represents the number of units which is given for a mass instance. Numbers of non-mass instances are represented by a special **set** notation which is stored in the set–table.

After having described the terms which are stored in the KB the tables which also belong to the Factual Knowledge shoul be explained. There are two tables which are used by the KB: the set–table and the equal–table.

The set–table contains information about each item that occured in a plural form. Plural forms are the following:

- simple plural expressions like *A truck transports computers.*
- plural expressions with numbers like *A truck transports 80 computers.*
- conjoined expressions like *A truck transports a computer and 15 disks.*

Most new entries in the set–table are inserted by the DM, but with respect to the treatment of conjoined terms, i.e. noun phrases which are combined by the conjunction *and*, the KB adds and removes entries of the set–table as well. The exact format of the set–table and the predicate which stores an entry of the set–table will be described by the DM- and TP-documentation because the corresponding teams have invented this representation.

The equal–table consists of entries which are stored by the predicate `ta_d_equal_table`. An entry combines two objects which are equalized by an update. An example will clarify this:

(5) The update *John is a driver* leads to a new entry in the equal–table which equalizes the object–terms for *John* and *driver* and which looks like `ta_d_equal_table( obj(112,[1,_,1,0], human,john), obj(113,[1,_,1,0],driver,_) )`.

Chart knowledge The Chart Knowledge consists of information about charts as a means of representation. This information includes a special identifier of a chart, the topic of a chart and some pointers to the terms stored in the KB and representing the knowledge which is made visible by drawing the chart. That means that the Chart Knowledge in itself does not include explicitly the information which is depicted by the chart. The Chart Knowledge is located in a table whose entries are stored by the predicate `ta_chart_table` which has 5 arguments. Further information about the representation of the Chart Knowledge is given in Deliverable T5.8 (subsection 2.3 The KB Representation of Chart Information).

Conceptual knowledge The Conceptual Knowledge is knowledge which is included in the system before any user-system interaction starts. This knowledge provides information about the concepts which is necessary to draw inferences triggered by user-system interactions. Conceptual Knowledge is represented by two means of representation: frames and rules.

An elaborated description of the frames and the knowledge they contain is given in the deliverables T5.4 (A Frame–Extension of DRSs for Supporting Conceptual Reasoning) and T5.5 (Augmentation and Modification of the Domain KB).

Rules contain Conceptual Knowledge which is not easily covered by frames. Primarily they deal with negated implications. In order to clarify this the contents of some rules are given as examples:

(6) If a thing is in or at a special place it is not in or at another place at the same time.

(7) If a thing is in or at a special place the thing is not moving towards a goal.

(8) If a driver drives a vehicle he cannot drive another vehicle at the same time.

(9) If an object arrives then it cannot depart at the same time.

These rules are implemented using the predicate `ta_f_ccr`. The heads represent the negated part of a rule and the bodies encode the if–part. The meaning is expressed by using Factual Knowledge terms and class-names which are known by the Conceptual Knowledge. More information about these rules are given in subsection 3.3.2.2 (Conceptual Contradictions) of Deliverable T5.8 (Elaboration of the KB Manager).

11.2.5 Direct Access to the Stored Knowledge

Working on the PROLOG level in the integrated or the stand–alone version of the KB, it is possible to have direct access to factual knowledge, chart knowledge, and conceptual

knowledge.

With respect to factual and chart knowledge the file **utililies** provides the following predicates which can be activated by typing:

lkb	lists all currently stored rel–terms, adj–terms, and comp–terms
lta	lists all current entries of the chart–table, the set–table, and the equal–table
dkb	deletes all terms which describe factual knowledge from the knowledge base and removes all entries of the chart–table, the set–table, and the equal–table; i.e. the knowledge base is removed from all factual and chart knowledge

The stored conceptual knowledge can be listed by using the predicate **show** whose code is placed in the file **show** and which has 5 arguments. Each argument can be a variable or can be instantiated by a value. The arguments relate to the structure of a frame so that a call of the predicate can be described by

```
show(<Framename>,<Slottype>,<Slotname>,<Aspect>,<Data>).
```

Each frame–part which matchs the given structure is listed in a structured way after calling the predicate **show**.

11.2.6 The Functions Concerning the Factual Knowledge

Factual Knowledge terms are delivered by the DM in order to be stored. This addition of new information to already stored Factual Knowledge is done by a special update–component which is placed in the KB. Factual Knowledge terms are also delivered to the KB in order to be confirmed or instantiated by already stored Factual Knowledge. The treatment of such queries is done by a KB–component which is called query–component. There are some other uses of already stored Factual Knowledge which are connected with special services for other components. They will be described in the chapter 'Relations to other ACORD components'. Now the functionality of the update–component will be presented and then a description of the query–component will follow.

Update–component For each term which is delivered to be updated, the update–component creates a fact of the predicate **ta_fact_update_term** whose only argument is the term to be updated. Then each fact of the predicate **ta_fact_update_term** is treated and after this the fact is removed. The contents of all facts of the predicate **ta_fact_update_term** denotes the terms which have to be updated but which aren't updated until now. Using this strategy it is easy to add new terms to be updated which are created during the update treatment of a term.

The first action which is done with the argument of a fact of `ta_fact_update_term` has to do with conjoined noun phrases: if there are noun phrases presented which includes an *and* they will be splitted so that for each item of the conjoined expression a special term is invented and the conjoined expression is removed. Then the terms are stored by the predicate `ta_fact_update_term` in order to be treated afterwards. The creation of new terms includes also adding and removing entries of the set–table. An example may clarify this:

(10) After splitting the conjoined expression *A truck transports a computer and 15 disks* the update is represented by two terms which contain a plural expression with a number and a simple plural expression, respectively. The terms correspond to the updates *A truck transports a computer* and *The truck transports 15 disks*.

This treatment means that each conjunction of noun phrases is considered as requiring a distributive reading which allows the splitting without changing the meaning. This splitting is done in order to faciliate finding the answer to queries which point to (only) one of the conjoined elements. Furthermore it allows a simple strategy to withdraw one element of a conjunction and to keep the other ones if this is necessary during consistency checking. It should be mentioned that no provision is made for collective readings. There is no algorithm available which determines whether a distributive or a collective reading is the intended one. Therefore the KB deals only with distributive readings.

The second step which is done during an update treatment of a Factual Knowledge term concerns non-numerical consistency. That means it is checked whether the new information is consistent with already existing factual information. This is done by negating the term which should be updated and then asking the query–component to verify this negated term. If the negated term is proved based on the already existing Factual Knowledge then an inconsistency would occur if the new term would be updated. In order to prevent this inconsistency and as it is assumed that the new information is more relevant than the old one, the already stored information which has led to a proof of the negated term has to be removed. An example may help to understand this strategy:

(11) Suppose the sentence *Truck1 is in Paris* was already updated and the new sentence to be updated is *Truck1 departs to Munich.* Then the new sentence to be updated is negated and the query–component is asked to verify *Truck1 does not depart to Munich.* This can be prooved by the conceptual consistency rule 'If a thing is in or at a special place the thing is not moving towards a goal'. In order to prevent an inconsistency between already stored information and the new term the old information *Truck1 is in Paris*, which verified the negated term, is removed.

Furthermore some consistency checking concerning numerical information is done. This kind of information is central for chart information and as charts play a major role in presenting information the consistency of numerical information is important. There are two different types of updates which concern numbers: number–defining information

and number–changing information. The first kind of information fixes a special number which makes obsolet each previous information given to the same topic. The second type of information changes a prior given information, i.e. 'old' information is still considered when treating this number–changing information. An example will help to understand the difference.

(12) The update *Truck1 transports 50 printers* is number–defining because prior given information concerning the number of transported printers is wiped out and furthermore not necessary to be considered. On the contrary an update like *Harry loads truck1 with 20 printers* is number–changing because the 'old' total number of transported trucks has to be taken into account in order to compute the new total number: the number consistency procedure adds the number given in the update to the 'old' total number and gets the new total number which replaces the 'old' total number.

Thus the number consistency checking is necesarry to assure that numbers are always describing the current state of the world, i.e. they provide common reasoning concerning numbers.

The update–component provides also some inferences which have to be drawn at update time because they influence the current state of the world. These implemented inferences are mainly connected with number consistency but there is a large variety of (not implemented) inferences which deal with non-numerical information. In the following an example of such an inference is given.

(13) Suppose the information *Truck1 is at the depot of Frankfurt* and *Truck1 contains 50 printers and 100 screens* is already stored. The update to be treated is *Harry empties truck1.* Then it is infered that each item of the load of truck1 is put in the depot of Frankfurt. This infered event is explicitly updated and some number consistency checking is done concerning the number of printers and screens stored in this depot. Furthermore it is concluded that eventually the truck transports nothing, i.e. the numerical information concerning the load of truck1 has to be removed.

Query component The query–component gets terms which should be confirmed (and instantiated if they contain variables) based on the stored factual knowledge. The general strategy is to treat each query term seperatedly. Then each factual knowledge term which belong to the same term class as the query term is analyzed whether it can confirm (or instantiate a variable of) the query term. It is not sufficient to stop the investigation of factual knowledge terms if a term is found which confirms (and instantiates variables of) the query term because further search may lead to information that also confirms (and instantiates variables of) the query term.

(14) Assume the factual knowledge includes terms related to the updates *Truck1 transports a computer* and *Truck1 transports 50 cables.* Given a query term representing the question *What does truck1 transport?* it is not sufficient to stop the search procedure after the first term which matches with *truck1*, which describes a transport event, and which instantiates the direct–object variable, i.e. after finding the term representing *Truck1 transports a computer.* If the search would be ceased the other relevant information, *Truck1 transports 50 cables*, would not be considered and the answer would be incomplete.

There are different inference strategies which are used by the query–component to find out whether a special factual knowledge term confirms (and instantiates variables of) the query term. These inferences can be devided in three classes: the direct–matching strategy, strategies which uses conceptual knowledge, and special strategies which depends on the query term and its contents. The application of the inferences is ordered from the simplest to the most complex strategy. If an inference succeeds, no further strategy is applied because the special factual knowledge term is recognized as confirming the query term and that is the goal of the query–component's activity. In the following the three mentioned strategies will be described. Further and more detailed information concerning the implementation of these different inference strategies are given in the deliverable T5.7 "First Implemantation of the Knowledge Base Manager" whose topic is exactly the application and implementation of these inferences.

The direct–matching strategy is very simple — but also very effective. The normal PROLOG matching strategy is used in order to test whether the query term matches the special factual knowledge term which is currently investigated. If the matching succeeds the factual knowledge term confirms the query term and moreover all possible variable instantiations are done. An example will show the applicability of this strategy.

(15) Suppose the factual knowledge (adjunct) term `ta_kb_af_adj( [spat(loc(in))], obj(100,[1,_,1,0],truck,truck1), obj(101,[1,_,1,0],city,paris))` representing *Truck is in Paris* is under consideration in order to confirm the query (adjunct) term `ta_kb_af_adj( [spat(loc(in))], obj(100,[1,_,1,0],truck,truck1), X)` for the question *Where is truck1?*. Direct matching leads to the instantiation of the variable `X` which stands for the location of *truck1*.

Strategies which use conceptual knowledge are activated if a the direct match fails. These strategies try to use conceptual knowledge in order to show that the items of the terms (i.e. object terms) which are not equal are different descriptions of the same thing (in the case of non–events) or concept based implications (in the case of events). The following examples will illustrate this.

(16) Suppose a term representing *Truck1 transports a computer* is updated and the user asks *Does a vehicle transport a computer?*. The non–event object terms for *truck1* and *a vehicle* are not matchable but the conceptual knowledge provides the query–component with the information that **vehicle** is a generalization of **truck**. Therefore the term representing *Truck1 transports a computer* can be used to instantiate the variable of the query term.

(17) Suppose a term representing *Truck1 departs from Paris* is updated and the user asks *Does truck1 go?*. The query–component comes to know by the conceptual knowledge that **depart** implies **go** and therefore uses the stored term to confirm the query.

The conceptual knowledge provides the query component with several kind of knowledge which makes this strategy very useful. A description of the represented conceptual knowledge is given in the deliverable T5.5 "Augmentation and Modification of the Domain KB".

The third class of strategies covers a wide range of specialized inferences. Some inferences are based on syntactical structures, the others are linked to common sense reasoning. At first, two examples for syntactically based special inferences are given.

(18) The fact that a special city is the location of a depot can be expressed by different linguistic means which leads to different syntactic structures: one can say *Paris has a depot* (represented by a rel–term) or *A depot is in Paris* (represented by a location adj–term) or one can use the genetive expression *the depot of Paris* (represented by a possessive adj–term). In order to answer the question *Does Paris have a depot?* the query–component therefore has to take into account each of the described syntactic structures; in this case it is not enough to look only at terms which are syntactically equal.

(19) Suppose the factual knowledge consists of the representations of the user inputs *Harry drives truck1* and *Harry is in Paris*. Now the query–component gets the representation of the question *Is a driver in Paris?*. In order to find out that the representation of *Harry is in Paris* confirms the question the query–component has to proove that Harry is a driver. This proof is done by using a special inference which takes into account appropriate conceptual knowledge. In this case the conceptual knowledge provides the query component with the information that a person can be called *driver* if the person is the subject of a driving event. Then the inference strategy tries to verify that Harry drives something. This can be done using the direct matching strategy. The described inference which prooves that Harry is a driver is syntactically based because it fullfills the task to verify a given description by examining syntactically linked representations.

There are some other special inferences which perform common sense reasoning. Especially spatial reasoning is supported by the implemented inferences. Therefore an example from this area should be given.

(20) Assume the representations of the updates *Truck1 is in Paris* and *Harry unloads truck1* are stored in the factual knowledge. In order to answer the question *Where is Harry?* the query–component uses the following inference strategy: An arbitrary locational information, e.g. the representation for *Truck1 is in Paris*, is taken and it is checked whether it can be proven that *truck1* and *Harry* are at the same place. This is the case if both items are involved in an event which requires that both participants are at the same place. The conceptual knowledge provides the query–component with the information that an unloading–event, which is stored in the factual knowledge, fulfils this requirement. So it is concluded that the location of *Harry* is the same as the location of *truck1*, i.e. that the location of *Harry* is *Paris*.

11.2.7 The Functions Concerning the Chart Knowledge

As described previously Chart Knowledge consists of knoweldge about the chart as a means of representation. But the knowledge about each chart involves also pointers to the knowledge which is depicted by a chart. There are predicates which allow the introduction of new charts and new knowledge delivered by charts, and which allow the changing and removing of such information. There is also a predicate which provides reading access to the Chart Knowledge and the knowledge which is depicted by a chart. These predicates are described in detail in the subsection 2.4 (Access to Chart Information) of the deliverable T5.8 (Elaboration of the KB Manager).

The part of the KB which treats updates presented by natural language creates for each new truck a chart showing the load of the truck and for each new depot a chart depicting the storage.

11.2.8 The Functions Concerning the Conceptual Knowledge

The Conceptual Knowledge is aquired before any user–system interaction starts. A special program is available which allows the addition, deletion, and changing of Conceptual Knowledge stored in frames. A detailed description about the aquisition of Conceptual Knowledge stored in frames can be found in chapter 2 (Components Relevant for Knowledge Aquisition) of deliverable T5.6 (Description of the KB–Components provided by TA).

Run–time access to the Conceptual Knowledge which is provided by frames is given by the predicate `ta_anfragen_mem` which has 6 arguments

```
ta_anfragen_mem( <Framename>,
                 <Slottype>,
                 <Slotname>,
                 <Aspect>,
                 <One-Element-Of-Data-List>,
                 <All-Data> )
```

If this predicate is called only the last argument has to be a variable: this argument will be instantiated with all the data that was found for the given information. The other arguments may be variables or instantiations; in the case they are variables they will be instantiated with appropriate frame information, otherwise the predicate fails.

There are no provisions to facilitate the aquisition of conceptual rules. The access to rules is done by calling the predicate `ta_f_ccr` with the Factual Knowledge term that should be proved.

11.3 Relations to Other ACORD Components

11.3.1 Commonly Used Tables

There is one table, the set–table, which is used by the KB, the DM and the TP. As described in section 2.1.3 "The Different Knowledge Types and Their Representation" the set–table represents the sets which are refered to by plural expressions. Mostly the DM puts new entries in the table because it gets new plural expressions directly from the parsers and transforms the InL-representation to the drs–representation. During the treatment of conjoined expressions, i.e. noun–phrases which are combined by *and*, the KB adds also new entries in the set–table. The KB and the TP make use of these entries in determining the number of things and in answering questions. Further description of the use of the set–table in order to answer *How–many*–questions is given in the documentation of the TP.

11.3.2 Cooperation with the Dialogue Manager

The DM delivers the terms to be updated to the update–component. In order to do this, the DM calls the predicate `ta_update_term_list`. This predicate has two arguments: the first argument is the list of terms to be updated and the second argument is used to return the chart identifier of those charts whose contents are concerned by the new updates.

Furthermore each contact to other components except the TP goes via the DM. There are interface predicates defined which allow the transmission of information combined with requests to find out something. In the following sections these interface predicates

are described.

11.3.3 Cooperation with the Theorem Prover

The TP gets each query which represents a user question and monitors the process of finding the correct answer(s). During this activity the TP asks the KB's query–component to verify single terms. The predicates which is provided by the KB's query–component to perform this is called `ta_verify`. It has three arguments: the term to be verified, a list with the verified term if the verification succeeds (otherwise it is an empty list), and a list with all terms which were used by the query–component to verify the term.

In order to prove the delivered terms the TP may have to compare several terms with different objects. The comparison of objects may lead to the problem that simple matching does not succeed and that it has to be checked whether the object terms are different, but acceptable descriptions of the same thing. This check is accomplished by some predicates of the query–component and described in the paragraph about the capabilities of the query–component. The KB provides some predicates which allow access to these checking predicates. The names of these predicates are `ta_comp_class` and `ta_comp_obj`.

There is also another cooperation with the TP. In the case of updating a new term the update–component performs some consistency checks for which the TP is asked whether it can proove the negation of a term to be updated by the use of negated terms which are directly stored in the TP. The predicate which does this has the name `tp_verify_negated_term`. It succeeds if the negated term can be prooven, otherwise it fails. In the case of success the predicate returns pointers to the negated facts which allowed the verification. Furthermore the TP provides the predicate `tp_remove_negated_term` which removes the given negated term stored in the data base of the TP. This predicate will also be called by the KB during the process of consistency maintenance.

11.3.4 Cooperation with the Resolver

The resolver is a component of ACORD whose input are InL–structures delivered directly by the parsers and whose output is given to the DM to be transformed to drs–terms. There is a close relationship between resolver and KBto perform the resolving task. The resolver takes the lead and makes requests to the KB. There are three different types of requests which are directed to the KB. They will be described in detail in the following.

The first kind concerns the request for some information stemming from conceptual knowledge. In order to find out the referent of definite descriptions the resolver asks the KB to proove the validity of a conceptual relation linking different descriptions. Using the query–predicate `ta_verify` the resolver gives terms to be verified by the conceptual

knowledge which states special conceptual relations between given descriptions. Then the query–component treats this query in the same way as each other query: It tries to proove the conceptual relation and returns the result to the resolver. An example will clarify this.

(21) Suppose the sentence *A truck is in Paris* is updated and the very next update is the sentence *Harry drives the vehicle.* The task of the resolver is to find out the referent of *the vehicle.* In order to do this the resolver asks the KB whether one of the following conceptual relations is valid between the classes `vehicle` and `truck`: specialization, generalization, participant relation. (The meaning of these relations is described in the deliverable T5.5 "Augmentation and Modification of the Domain KB.) The call to verify the generalization relation looks like the following: `ta_verify( generalize(vehicle,truck), Result_List, [] ).`

The second kind of collaboration also concerns the resolution of definite descriptions. But now the requests don't concern conceptual relations between classes but are directed directly to find an instance which is already known as a part of the factual knowledge and which fulfils given requirements. In order to do this the resolver builds a query whose structure corresponds to the representation of a normal user question. Therefore the query–component of the KB performs its work without knowing whether the query comes from the user or the resolver. The following example shows the importance of this kind of collaboration.

(22) Suppose that at some time the user has stated *Truck1 is in Paris.* Then the user gives the information *Harry drives the truck which is in Paris.* Now the resolver has to find out which truck is refered to by the expression *the truck which is in Paris.* As the KB stores the factual knowledge which has to be accessed to find the referent, a query is sent to the KB using the predicate `ta_verify`. Then the query–component treats this query which corresponds to the question *Which truck is in Paris?* and instantiates the query with the object term representing *truck1.* Then this information is returned to the resolver which replaces the representation of the description *the truck which is in Paris* by the representation of *truck1.*

The third kind of collaboration has to do with desambiguation of the parser output. With respect to some syntactic structures and some lexical items a set of representations may be found by the parser and some semantic knowledge (especially concerning conceptual knowledge) is necessary to find an interpretation which is also semantically acceptable. The resolver is responsible to desambiguate such representations. This is done by delivering a possible (rel– or adj–)term to the KB and to ask whether this term fulfils semantic constraints which are stored in the conceptual knowledge. This checking is performed by the predicates `dm_check_pred_frame` (checking rel–terms) and `dm_check_adjunct_frame` (checking adj–terms). An example will show the necessity of these checks.

(23) The user input *Harry unloads a pc* leads to the syntactic ambiguity that the representation of *a pc* is considered either as the direct object (the thing which serves as a receptacle) or as the indirect object (the thing which is put at another place). The KB rejects the rel–term which contains the representation of *a pc* as direct object and verifies a rel–term which contains this representation as indirect object based on conceptual knowledge. The conceptual knowledge provides the information that the direct object of an unloading event has to belong to the class `means_of_transport` or `receptacle` and that the indirect object has to belong to the class `goods`.

11.3.5 Cooperation with the Visualization Component

One main topic of ACORD is the integration of natural language and graphics. There is some information that may be given in natural language and which is visualizable. The following enumeration lists the different kinds of visualizable information.

cities having a depot	expressed by changing the shape of the city on the map
trucks which are located in or at a city	expressed by placing the icon of a truck near the city the truck is
trucks which are moving from and/or towards a city	expressed by trucks whith rotating wheels and by placing the truck between the 'source' and the 'goal'

The update–component of the KB checks each update whether it contains new or changes prior given visualizable information. If this is the case the update–component sends factual knowledge terms to the visualization component which express the visualizable information. This is done by using the predicate `ta_update_term_list`: At the call of this predicate one argument is a variable and the update–component instantiates this argument with a list containing the factual knowledge terms to be visualized.

Furthermore the update–component reacts if information is given which concerns charts. On the one hand each introduction of a truck or a depot leads to a hint for the visualization component to draw a new chart for the item. On the other hand each update is checked whether it contains information that touches the contents of an existing chart. As the KB knows what is depicted by each chart it can give a hint to the visualization component if a new update changes the contents of a chart. Then the visualization component reacts by demanding the new information to be depicted by the chart and by redrawing it. The hints concerning charts are given by delivering the numbers of the newly invented or changed charts which are put in a list. This list is the instantiation of an argument of the predicate `ta_update_term_list` which is called to introduce a new update.

11.3.6 Cooperation with the Generator

One task of the GENERATOR is the generation of natural language descriptions for objects. Normally the identifier, that is a number, and the class is given as starting point for the generation. If there are other objects known which belong to the same class a description consisting only of the given class is ambiguous. Therefore the GENERATOR has to produce a unique (definite) description based on the knowledge the user and the system have about the object. As the KB stores all (non–negated) factual knowledge it can help to find a definite description provided there is one. The KB offers the predicate `ta_make_unique_descr` which gets the identifier number of the object under consideration and tries to return a set of terms which gives a unique definite description. The strategy of this predicate is to take each term which contains the object and to check whether the represented information is also valid for other objects or not. If this is not the case the checked term can be considered as a unique definite description.

11.4 Potential of Development

The KB is a very central component of ACORD and its functionality depends very much on the work of the other components. Nevertheless there are some features and parts of the KB which can be used independently from ACORD or which provide solutions on the implementational level which can be used in other systems similar to ACORD.

The first to mention is the formalism which was invented to represent conceptual knowledge. This formalism is already used in other projects in order to store conceptual knowledge of a very different domain: the features of a special operation system. Combined with the predicates to aquire, test, and change conceptual knowledge this module of the KB is well–developed and is a useful tool for other projects.

Connected to the representation of conceptual knowledge it is worth to mention that it is easy to augment the coverage of conceptual knowledge, which is now limited to the transport domain. Here lays a great potentiality of development.

Furthermore some features and procedures of the KB can be used in other knowledge based systems because they treat frequent problems and tasks. The implemented solutions to problems concerning inference strategies, consistency checking, and finding definite descriptions are among them.

Finally it should be said that the code of the KB, which is written in PROLOG, can be translated in another programming language. The only thing to be alert of is that the functionalities of PROLOG, like backtracking and unification, are also provided.

Chapter 12

UCG grammars ; the control of their descriptive adequacy

Gabriel G. Bès,[1]
Pierre-François Jurie
CLF

12.1 Scientific Background and Development

Several models of grammars were used in the ACORD project (LFG, UCG, [illegible], string grammars and CCG). Only the first two remained as the standard grammar models of the project. Furthermore, the UCG model can be understood as a general framework leaving open choices : these are represented in the English and French grammars in their several and evolving versions (see 3 and 4, but also in other UCG epigona (cf Bès et Gardent 89, Gardent et al. 89, Popowich 89, Sanfilippo 90, and Beaven 90).

The justification of the choice of a particular model has been an important concern of the ACORD project ; Deliverable T1.1 and Deliverable T1.2 discuss about grammar and semantic models respectively. The question of how to choose between particular UCG grammars was a major issue discussed in Clermont-Ferrand parsing meeting of September 87. In general, discussions about choices of formalisms consumed considerable amount of time and energy (see Jurie 88). Furthermore, reviewers of the project and participants to conferences where theACORD project was presented, always formulated the same question: why two different models of grammars are in the same project ? Is it possible to justify them otherwise than by social pressure and/or academic tradition ?

[1] Alphabetical order

No answer has ever been given to these questions. The issue about the criteria of choice of alternative UCG grammars, which manifest a foreseen and confirmed trend to proliferation, remains opened.

It is in this background that the work on the control of grammars grew. This work has BOTH theoretical (see the following) AND practical goals (see 12.4). It was developped in the last part of the project.

Taken for granted that (computational) linguistics is an empirical science, (computational) hypothesis, expressed in the form of grammars, must be checked and validated with observations and data. Although this point seems uncontroversial, little has been done to capture observations in a formal way theoretically neutral with respect more or less explicit choices introduced in particular models of grammars. The work on control of grammars attempts to overcome this situation: it pretends to formulate in a compact way natural language observations, and to check the adequacy of a particular grammar by a calculus on it, in order to evaluate its descriptive adequacy. Though descriptive adequacy is not the unique parameter with respect to which a particular grammar must be selected, it is worthwhile to recall that (computational) linguistics being empirical, no rational for choosing among competing formalisms exists without it.

12.2 Technical description

The control of grammars consists in three parts: (i) a descriptive metalanguage, specifying the set GS of grammtical sentences; (ii) a calculus on grammar, specifying the set FS of formalised sentences; (iii) a comparison of GS and FS .

A descriptive metalanguage (for short **metalanguage**) is a formalised knowledge about the object language but it differs from the particular grammar models associated with the same object language.

Besides formal properties, we suggest that a grammar, when referring to such objects as LFG UCG GPSG, FUG, HPSG particular grammars are intended to satisfy at least two basic requirements

- association to parsing and generation
- characterisation of NL

By the first of these requirements, we understand that the grammars must be integrated in parsing and generation mechanisms in efficient ways. For fulfilling this requirement,

the models of grammars use "constructive" rules, that is, rules which are not only inferencing rules, but which, in a hidden way, condition the algorithmic processes of parsing and generation.

None of the above is a basic requirement for a metalanguage. A metalanguage is not intended to say something clever about NL. The notions of "insight" and "efficiency", the latter one related to parsing and generation, are definitely alien to it. It specifies grammatical sentences, but it leaves opened what the significant generalization are and it introduces no choice on algorithmes which can be designed to analyse or generate effectively the specified sentences.

The metalanguage consists in two parts

i A formalisation of the lexicon

(a) A finite set VAL

(b) A relational structure in a first order language

LEX = $< E, C_0...C_n, P_0...P_n, R_0...R_n, ... >$

which is a model of

a set of axioms T1

[E denotes a set : the set of formalised lexical signs, C_0, C_1, denote individual constants, $P_0...P_n$ denote unary predicates, $B_0...B_n$ binary predicates ...]

ii A formalisation of the sentence

A formalised sentence is a pair

$< X, Cons >$

where

(a) X is a string

$X =< X_0, ..., X_n >$

-either an element of E

-or a formalised sentence

(b) *Cons* is a binary predicate such that $Cons(i,t) \Rightarrow n \geq i$, and $t \in$ VAL

[$Cons(i,t)$ means that the sign which is in the place number i, is interpreted as being in the relation named by t]

Example :

VAL = {nom, obj, ...}

Individual constants : *ne, pas, que, ...*

Unary predicates ; *verb, np, cln, wh, lex, ...*

T1

- The predicates *verb, cln, wh, lex* are disjoint
- $np = Wh \cup lex$

...

T2

1. ∃i verb(xi)
2. verb(i) & verb(j) ⇒ i = j
3. cons(i, nom) & cons(j, nom) & np(i) & np(j) ⇒ i = j
4. cons(i, obj) & cons(j, obj) ⇒ i = j
5. cons(i, nom) ⇒ cln(xi) or np(xi)
6. cons(i, obj) ⇒ np(xi)
7. cln(xi) & cln(xj) ⇒ i = j
8. ∃j cons(j, nom)
9. verb(xi) & clnom(xj) ⇒ j = i + 1
10. verb(xi) & cons(k, nom) & k < i ⇒ i = k + 1
11. verb(xi) & cons(k, nom) & i < k ⇒ k = i + 1
12. verb(xi) & cons(k, nom) & np(xk) & i < k ⇒~ ∃j clnom(xj) & lex(xk)
13. verb(xi) & clnom(xj) & i < k ⇒~wh(xk)
14. verb(xi) & clnom(xj) & k < i & cons(k, nom) ⇒~wh(xk)
15. verb(xi) & cons(k, nom) & np(xk) & i < k ⇒ ∃j[wh(xj) & j < k]

Axioms	Paraphrasis
(1) + (2)	There exists exactly one place in the sentence which is occupied by a verb
(3)	There exists at most one nominative place occupied by a *np*
(5)	Only nominative clitics (=cln) or *np* can occupy a nominative place
(4) + (6)	There exists at most one obj-place and this place (if it exists) must be occupied by a *np*
(7)	There is at most one place which can be occupied by a cln
(8)	There is at least one nominative place
(9)	The place of cln (cf. it exists) is just after the place of the verb

Axioms	Paraphrasis
(10) + (11)	Nominative places are just next to the verb
(13) + (14)	If there exists a place for cln (inversion 1) then no place after the verb and no nominative place before the verb can be occupied by a wh
(12) + (15)	If a nominative place is after the verb and is occupied by a np (inversion 2) then there is no place for a nominative clitic, the np in the nominative place is *lex* and there exists a place before the verb which is occupied by a *wh*

The calculus on grammar is founded on the notion of **closure table**, defined in the following.

Definition. A closure table is of the form:

$R_1 A_1 B_1 C_1$

$R_2 A_2 B_2 C_2$

...

$R_n A_n B_n C_n$

where the following conditions are satisfied

(a) A_i, B_i, C_i are templates (Categories + features)

(b) R_i is a binary rule

(c) if A_i is a template of a sign X, and

B_i is a template of a sign Y

then there exists a sign Z such that:

X + Y $\stackrel{R_i}{\Rightarrow}$ Z and C_i is a template of Z

(d) if

X + Y $\stackrel{R}{\Rightarrow}$ Z

then there exists i such that

$R_i = R$

A_i is a template of X

B_i is a template of Y and

C_i is a template of Z

The closure table can be used immediately to determine the set of all the strings $X_1 ... X_n$ of lexical signs such that

$$X_1 + \ldots + X_n \overset{GR}{\Rightarrow} sent$$

More precisely
definition A closure tree is a tree in which

(a) the root is the template ***sent***

(b) every vertex C has two sons A and B and there exists a binary rule R such that

RABC

is a line of the closure table (a straightforward extension is needed for unary rules)

(c) every leaf is lexical

Result

(a) the set of all the closure trees can be automatically founded from the closure table, and:

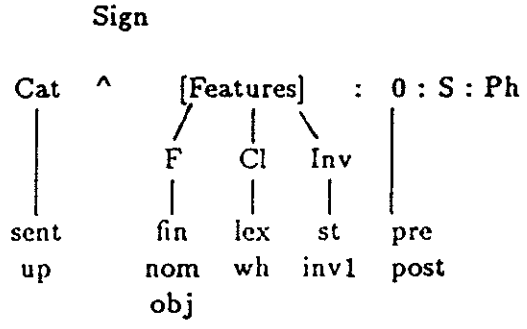
(b) $X_1 + \ldots + X_n \overset{GR}{\rightarrow} sent$

iff

there exists a closure tree which string of leaves is $A_1, \ldots, A_n$ and for every i A_i is a template of X_1

For all presently examined UCG grammars it is in principle possible to construct a closure table. Furthermore, we conjecture that, given the formal properties of UCG the construction of such a table is always possible. The conjecture is based in particular on the fact that UCG grammars (like pure categorical grammars) generate only a finite number of syntactically different signs from a finite number of entries. For grammars which use more intensively type-raising and computational rules, for instance, the situation may be completely different.

Example. **The Grammar G :**



Entries :

V : sent^ [...st]/(np^[nom...]: pre)/(np^[obj...]: post: ...
NP : C/(C/(np^[nom or obj, X ...]: O): O):...
Cln sent^[...inv1]/(np^[nom,lex...]: pre)
/(np^[obj, lex...]: post)
/(sent^[...st]/(np^[nom...]: pre)
/(np^[obj...]: post) : post)

Rules :

FA X/Y: ...: a Y: ...: b ⇒ X: ...: ab
BA Y: ...: a X/Y: ...: b ⇒ X: ...: ab

Notation :

S, Ph: omitted
...: underspecification
V inv1: sent^[...inv1] / (np^[nom, lex...]: pre)
/ (np^[obj, lex...]: post)
lex-NP, wh-NP: NP with X instantiated by *lex* and *wh* respectively

Closure table of G

N°	Rule	Functor	Argument	Result
1	BA	NP	V	sent/(np^[nom...]: pre)
2	FA	NP	sent^[...st]/(np^ [nom...]: pre)	sent
3	BA	Cln	V	Vinv1
4	BA	lex-NP	Vinv1	sent/(*np*^[nom,lex...]: pre)
5	FA	lex-NP	sent^[...inv1]/(np^[nom,lex...]: pre)	sent

The table is constructed along a chain of successive extensions:

- Lines 1 and 2 form a closure table for the subgrammar G1 of G which has only entries V and NP
- Lines 1 to 5 is a closure table for the grammar G = G1 + Cln

The comparison of GS and FS proceeds by induction on the length of the strings, and examination of particular cases (case disjunction method of proof). In the above example, it can be easily proved that G is sound but not complete. Our proofs strongly depend on the features of the particular grammar studied, but we are studying at present the possibility of giving a more uniform method of proof.

12.3 Relations to other ACORD components

The methodology developped in relation to the grammar can be applied in principle to other ACORD components and particularly to the control of the Resolver and the KB conceptual knowledge.

12.4 Potential of development

Today nobody knows explicitely what the coverage of a parser or a generator is. Demos and current specifications are, at the best, hints indicating roughly what the system can or must perform. When a lot of money is spent in the implementation of grammars and parsers, it is a big disppointment to find in a haphazard way evident but completely unforeseen counter-examples to some system. The point is that when some unexpected counter-examples are established, the reliability of the whole system decreases dramatically. And reliability is *sine qua non* requirement of any technological product: we are not aware of any long-lasting technological market founded on the marketing of an unreliable product. Control of grammrs can add reliability to natural language systems.

With an explicit metalanguage it will be possible to specify accurately corpus of linguistic observations and, by the way, to put in perspective achievements of particular models of grammars, evaluating thus what they can effectively do. Control of grammars is thus an important step toward the elaboration of NL standards and grammatical models evaluation and comparison.

Currently particular grammars overgenerate. On the other hand, any non-exclusively natural language system must be prepared to deal with a corrupt input, i.e. conditions on grammaticality must be relaxed in specified situation. It is difficult to foreseen how this can be done if the effective descriptive adequacy of particular grammar remains obscur. Control of grammars looks like an unescapable tool for robust natural language understanding.

References

Adobe Systems inc. [1987] *Postscript Language Reference Manual.* Addison-Wesley.

Aho, A.V., Sethi, R., and Ullman, J.D. [1986] *Compilers - Principles, Techniques, and Tools.* Addison Wesley, 1986.

Beaven, J.L. [1990] A Unification Based Treatment of Spanish Clitics. In *Proceedings of the DYANA Workshop on Word order in Categorial Grammar*, Université Blaise Pascal, Clermont-Ferrand, 25-27 May 1990.

Bès, G.G. and Gardent, C. [1989] French order without order. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester Institute of Science and Technology, Manchester, UK, 10-12 April, 1989, pp. 249-255.

Beth, E. W. [1959] *The Foundation of Mathematics.* North Holland, 1959.

Boolos, G. [1984] Trees and finite satisfiability. In *Notre Dame Journal of Formal Logic.* 1984.

Bijl, A. [1985] *Boundaries and Interfaces between Components.* Position Paper. EdCAAD, 1985.

Bijl, A. [1987] *Graphics as Language.* Position Paper. EdCAAD, 1987.

Bijl, A. [1988] *Semantics of Graphics, An Introduction to* ACORD *Deliverables T3.4 and T3.6.* EdCAAD, 1988.

Bresnan, J. [1982] ed. *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts, 1982.

Calder, J., Klein, E. and Zeevat, H. [1988] Unification Categorial Grammar: A Concise, Extendable Grammar for Natural Language Processing. In *Proceedings of the 12th International Conference on Computational Linguistics and the 24th Annual. Meeting of the Association for Computational Linguistics* Budapest, 22-27 August, 1988.

Calder, J., Reape, M. and Zeevat, H. [1989] An algorithm for generation in Unification Categorial Grammar. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics* Manchester, 10-12 April, 1989, pp 233-240.

Cardelli, L. [1984] A Semantics of Multiple Inheritance. In Kahn, G., MacQueen, D. B. and Plotkin, G. (eds.) *Semantics of Data Types*, Institut National de Recherche en Informatique et en Automatique Centre de Sophia-Antipolis, Valbonne, June 1984, 1984, pp 51-68.

Clocksin, W. F. and Mellish, C. S. [1981] *Programming in Prolog.* Springer-Verlag, Berlin.

Dörre, M. and Momma, S. [1987] Generierung aus f-Strukturen als strukturgesteuerte Ableitung. In *Proceedings of GWAI-87* Geseke, October 1987.

Eberle, K. and Kasper, W. [1989] Tenses as Anaphora. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics.* Manchester, 10-12 April, 1989.

Eisele, A. [1987]. *Eine Implementierung rekursiver Merkmalstrukturen mit disjunktiven Angaben.* Diplomarbeit, Institut f. Informatik, Stuttgart.

Eisele, A. and Dörre, J. [1986] A Lexical Functional Grammar System in Prolog. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, 25-29 August, 1989.

Eisele, A. and Dörre, J. [1988] Unification of disjunctive feature descriptions. In *Proceedings of the 26rd Annual Meeting of the Association for Computational Linguistics* Buffalo, NY.

Eisele, A. and Schimpf, S. [1987] *Eine benutzerfreundliche Softwareumgebung zur Entwicklung von LFGen.* Studienarbeit, Institut für Informatik, Stuttgart.

Enderle, G., Kansy, K. and Pfaff, G. [1984] *Computer Graphics Programming.* Springer-Verlag.

Fenstad, J.E. et al. [1987] *Situations, Language and Logic.* Reidel: Dordrecht.

Gardent, C., Bès G.G., Jurie, P.F. and Baschung, K. [1989] Efficient Parsing for French. In *Proceedings of the Conference of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, BC, Canada, 26-29 June, 1989, pp. 280-287.

Grosz, B. [1977] The representation and use of focus in a system for understanding dialogs. In *Proceedings of the fifth International Joint Conference on Artificial Intelligence.* Cambridge, 1977.

Grosz, B. and Sidner, C. L. [1986] Attention, intentions, and the structure of discourse. In *Computational Linguistics.* 1986.

Halvorsen, P-K. [1987] *Situation Semantics and Semantic Interpretation in Constraint-based Grammars.* CSLI-Report CSLI-87-101, Stanford, 1987.

Halvorsen, P-K., and Kaplan, R. [1988] Projections and Semantic Description in LFG. In *Proceedings of the International Conference on Fifth Generation Computer Systems* Tokyo, 1988.

Johnson, M. E. [1987] *Attribute-Value Logic and the Theory of Grammar.* PhD Thesis, Department of Linguistics, Stanford University, 1987.

Jurie, P.F. [1988] *INL and DRS Formalisms Compared,* University of Clermont-Ferrand, 1988.

Kalish, D. and Montague, R. [1964] *Logic.* 1964.

Kamp, H. [1981] A Theory of Truth and Semantic Representation. In Groenendijk, J.A. et. al. (eds.) *Formal Semantics in the Study of Natural Language* Vol. I, Amsterdam, 1981.

Kaplan, R. [1987] Three seductions of computational psycholinguistics. In Whitelock, P. et al. *Linguistic Theory and Computer Applications.* London, 1987, pp 149-188.

Kaplan, R. and Bresnan, J. [1982] Lexical-Functional Grammar : a formal system for grammatical representation. In Bresnan, J. (ed.) *The Mental Representation of Grammatical Relations.* Cambridge Mass., 1982, pp 173-281.

Kaplan, R. and Maxwell, J. [1988] An Algorithm for Functional Uncertainty. In *Proceedings of the 12th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Budapest, 22-27 August, 1988.

Karttunen, L. [1984] Features and Values. In *Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics,* St nford University, Stanford, Ca., 2-6 July, 1989, pp 28-33.

Karttunen, L. [1986] D-PATR: A Development Environment for Unification-Based Grammars. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, 25-29 August, 1986, pp 74-80.

Kasper, R. T. [1987] *Feature Structures: A Logical Theory with Application to Language Analysis.* PhD Thesis, Electrical Engineering and Computer Science Department,

University of Michigan.

Kasper, W. and Reinhardt, K. [1988]. *Anaphora Resolution for Discourse Representation Theory.* Ms, Stuttgart.

Kay, M. [1979] Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society*, 1979, pp 142-158.

Kohl, D. [1988] *Generierung funktionaler Strukturen aus einer semantischen Reprasentation.* Diplomarbeit, Institut für Informatik, Stuttgart.

Kress, G.R. [1976] ed. *Halliday: System and function in language.* Oxford: Oxford University Press.

Kuhn, T. S. [1962] The structure of scientific revolutions. In *International Encyclopedia of Unified Science, Vol. 2, No 2.* University of Chicago Press.

Lee, J. R. [1987] *Lodable C Functions for the* ACORD *Graphics Component.* Technical Paper. EdCAAD, 1987.

Lee, J. R., Bijl, A. and Szalapaj, P. J. [1986] *The Graphics Component of the* ACORD *System.* Position Paper. EdCAAD, 1986.

Lee, J. R., Kemp, R. and Manz, T. [1989] Knowledge-based graphical dialogue : A strategy and architecture. In *ESPRIT '89.*

Lee, J. R. and Szalapaj, P. J. [1986] *Prototype Graphics System : Specification.* ACORD Internal Deliverable. EdCAAD, 1986. **Mellish, C. S. [1988]** Implementing Systemic Classification by Unification. In *Computational Linguistics* **14.1**, pp 40-51.

Moens, M., Calder, J., Klein, E., Reape, M. and Zeevat, H. [1989] Expressing generalizations in unification-based grammar formalisms. In *Proceedings of the 4th European Meeting of the Association for Computational Linguistics* Manchester, 10-12 April, 1989, pp 174-181.

Momma, S. and Dörre, J. [1987] Generation from *f*-structures. In Klein, E., Benthem, J. v. (eds.) *Categories, Polymorphism and Unification*, Center for Cognitive Science, University of Edinburgh.

Netter, K. [1986] Getting Things out of order. (An LFG Proposal for the treatment of German Word Order). In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, 25-29 August, 1989.

Netter, K. [1987] Wortstellung und Verbalkomplex im Deutschen. In Klenk et al. *Computerlinguistik und Philologische Datenverarbeitung* Hildesheim, 1987.

Netter, K. [1988] Non-Local Dependencies and Infinitival Constructions in German. In Reyle, U. and Rohrer, C., *Natural Language Parsing and Linguistic Theory.* Dordrecht: Reidel.

Netter, K. and Rohrer, C. [1988] Syntactic Analysis in Lexical Functional Grammar. The Example of German Prepositional Phrases. T.A. Informations, 1988, 28.2, Numéro Spécial: Linguistique et Informatique en République Fédérale Allemande.

Pelletier, F. J. [1986] Seventy-five Problems for Testing ATP. In *Journal of Automated Reasoning.* 1986.

Pereira, F. C. N. and Shieber, S. M. [1984] The Semantics of Grammar Formalisms Seen as Computer Languages. In *Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual the Association for Computational Linguistics*, Stanford University, Stanford, Ca., 2-6 July, 1984, pp123-129.

Pollard, C. and Sag, I. [1987] *An Information-Based Approach to Syntax and Semantics: Volume 1 Fundamentals.* Stanford, Ca.: Center for the Study of Language and Information.

Popowich, F. [1989] Tree Unification Grammar. In *Proceedings of the Conference of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, BC, Canada, 26-29 June, 1989, pp. 228-236.

Reinhart, T. [1983] *Anaphora and Semantic Interpretation.* London 1983.

Reyle, U. [1988] Compositional Semantics for LFG. In Reyle, U. and Rohrer, C. (eds.), *Natural Language Parsing and Linguistic Theories.* Reidel: Dordrecht.

Reyle, U. and Frey, W. [1985] Grammatical Functions, Quantification and Discourse Referents. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 1985.

Sanfilippo, A. [1990] Inversion, Dislocation and the Null-Subject Parameter. In *Proceedings of the DYANA Workshop on Word order in Categorial Grammar*, Université Blaise Pascal, Clermont-Ferrand, 25-27 May 1990.

Shieber, S. M. [1986] *An Introduction to Unification-based Approaches to Grammar.* Chicago, Illinois: The University of Chicago Press.

Shieber, S., Uszkoreit, H., Pereira, F. C. N., Robinson, J. J. and Tyson, M. [1983] The Formalism and Implementation of PATR-II. In Grosz, B. and Stickel, M. E. (eds.) *Research on Interactive Acquisition and Use of Knowledge*, SRI International, Menlo Park, 1983, pp39-79.

Sidner, C. L. [1979] *Towards a Computational Theory of Definite Anaphora Comprehension in English Discourse.* MIT Artificial Intelligence Laboratory, 1979.

Siekman, J. H. [1975] *String-unification, part 1.* MS. Essex University.

Smullyan, [1968] *First Order Logic.* Berlin, 1968.

Tweed, C. [1985] *Dynamic Loading in C*-PROLOG. Technical Paper. EdCAAD.

Tweed, C. and Bijl, A. [1988] MOLE : a reasonable Logic for Design ? In Hagen, P. J. W., Tomiyama, T. and Akman, V. (eds.)*Intelligent CAD Systems 2 : Implementation Issues.* Berlin : Springer-Verlag.

Wedekind, J. [1986] A concept of derivation for LFG. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, 25-29 August 1989, pp. 486 - 489.

Wedekind, J. [1988] Generation as Structure driven Derivation. In *Proceedings of the 12th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics* Budapest, 22-27 August, 1988, pp 732-737.

Zeevat, H. [1986] *A Specification of InL.* Internal ACORD Report. Centre for Cognitive Science, Edinburgh, 1986.

Zeevat, H., Klein, E. and Calder, J. [1987] An Introduction to Unification Categorial Grammar. In Haddock, N. J., Klein, E. and Morrill, G. (eds.) *Edinburgh Working Papers in Cognitive Science,* Volume 1: *Categorial Grammar, Unification Grammar, and Parsing.*

Index

List of Deliverables

[T1.1]
Evaluation of Different Parsers and Syntactic Theories. ACORD Deliverable T1.1. BULL, IMS 1985.

[T1.2]
Kasper, W. *Montague Grammar, Situation Semantics, and Discourse Representation Theory. A Comparison of Three Semantic Theories.*ACORD Deliverable T1.2. IMS 1986.

[T1.2b]
Sedogbo, C. *Deductive Natural Language Processing for DRSs.* ACORD Deliverable T1.2. BULL 1986.

[T1.2c]
Sedogbo, C. *The Representation of Discourse structures in* PROLOG. ACORD Deliverable T1.2. BULL 1986.

[T1.3]
Sedogbo, C. *Deductive Natural Language Processing for DRSs.* ACORD Deliverable T1.3. BULL 1986.

[T1.4]
Netter, K., Karcher, U., Eisele, A. and Dorre, J. *Documentation of the German Grammar and STUT's Parser System.* ACORD Deliverable T1.4. IMS 198[illegible]

[T1.5]
Sedogbo, C. and Salkoff, M. *French Parser.* ACORD Deliverable T1.5. BULL 1986.

[T1.6]
Calder, J. Klein, E., Moens, M. and Reape, M. *General Constraints in Unification Grammar.* ACORD Deliverable T1.6. ECCS 1988.

[T1.7'(a)]
Zeevat, H. (ed) *Specification of the Central Pronoun Resolver.* ACORD Deliverable T1.7'(a). IMS 1988.

[T1.7'(b)]
Eberle, K., Eier, S., Gardent, C., Kasper, W., Reape, M., Tops and Zeevat, H. *Extension of the Anaphora Resolution.* ACORD Deliverable T1.7'(b). IMS, ECCS, CLF 1989.

[T1.8a]
Gabbay, D., Hoepelman, J., Machate, J., Valerius, R. and Hoof, T. v. *Investigation on the Deduction Component.* ACORD Deliverable T1.8. FhG 1986.

[T1.8b]
Ommani, F. and Sedogbo, C. *Principles and Implementation of a Theorem Prover Based on the Tableau Calculus.* ACORD Deliverable T1.8. BULL 1988.

[T1.10]
Bethke, Frey, Kamp and Zeevat, H. *Extension of the Theorem Prover to Intensional Phenomena.* ACORD Deliverable T1.10. IMS 1989.

[T1.11a]
Ommani, F. and Hoof, T. v. *Concept and Implementation of Plurals in the* ACORD *system.* ACORD Deliverable T1.11a. BULL 1989.

[T1.11b]
Ommani, F. and Hoof, T. v. *Expansion of Deduction Rules Operating on the Semantic Representation.* ACORD Deliverable T1.11b. BULL 1989.

[T1.12]
Calder, J. (ed.) *Polytheoretic Lexicons and Reusable Dictionaries.* ACORD Deliverable T1.12.ECCS 1989.

[T2.1]
Calder, J., Klein, E., Moens, M. and Zeevat, H. *Problems of Dialogue Parsing.* ACORD Deliverable T2.1. ECCS 1986.

[T2.2]
Baschung, K. and Bes, G.G. *Feasibility of a GPSs French Grammar.* ACORD Deliverable T2.2. CLF 1985.

[T2.3]
Baschung, K., Bes, G.G., Corluy, A., Guillotin, T., Panckhurst, R. and Zeevat, H. *Contextual Phenomena in Dialogue.* ACORD Deliverable T2.3. LdM, CLF and ECCS 1986.

[T2.4]
Steedman, M.J. *Incremental Interpretation of Dialogue.* ACORD Deliverable T2.4. ECCS 1986.

[T2.5]

Moens, M. and Steedman, M.J. *Temporal Information and Natural Language Processing.* ACORD Deliverable T2.5. ECCS 1986.

[T2.6]
Calder, J., Moens, M. and Zeevat, H. *A* UCG *Interpreter.* acord Deliverable T2.6. ECCS 1986.

[T2.7]
Baschung, K., Bes, G.G., Panckhurst, R., Coluy, A. and Guillotin, T. *Syntactic French Parser for Dialogue.* ACORD Deliverable T2.7. LdM and CLF 1987.

[T2.7'(a)]
Netter, K. *Syntactic Aspects of LFG-based Dialogue Parsing.* ACORD Deliverable T2.7'(a). IMS 1988.

[T2.10]
Kohl, D., Plainfossé, A., Reape, M., Gardent, C. *Text Generation from Semantic Representation.*ACORD Deliverable T2.10. ECCS, LdM, CLF,IMS 1989.

[T3.1]
Krishnamurti, R. and Sykes, P. *A Graphics Standard for* PROLOG*: The* PROLOG*/GKS Binding.* ACORD Deliverable T3.1. EdCAAD 1986.

[T3.2]
Lee, J. R. *Interactive Computer Graphics : A Literature Review and Position Paper.* ACORD Deliverable T3.2. EdCAAD 1985.

[T3.3]
Krishnamurti, R. and Kemp, R. *Implementation of* PROLOG*/GKS : An Overview of C-*PROLOG*/GKS.* ACORD Deliverable T3.3. EdCAAD 1987.

[T3.4]
Szalapaj, P. *A Syntax for a Drawing Machine.* ACORD Deliverable T3.4. EdCAAD 1987.

[T3.5]
Krishnamurti, R. *Issues in Shape Editing and Shape Representation.* ACORD Deliverable T3.5. EdCAAD 1987.

[T3.6a]
Krishnamurti, R. *Representational Semantics for Graphical Discourse.* ACORD Deliverable T3.6a. EdCAAD 1988.

[T3.6b]
Szalapaj, P., Lee, J. and Bijl, A. *The Semantics of Computational Systems that Depict Non-graphical Information.* ACORD Deliverable T3.6. EdCAAD 1988.

[T4.1a]
Hanne, K.H. *Report on the Investigation into the Literature on Combined Graphics Textual Systems.* ACORD Deliverable T4.1a. FhG 1986.

[T4.1b]
Hoepelman, J., Hoof,T. v. and Dellinger, W. *Review of Research on the Semantics of Deictic Phenomena and Natural Language* ACORD Deliverable T4.1b. FhG 1986.

[T4.1c]
Meulen, A.G.B.,t. and Bouma, G. *Research on the Semantics of Natural Language and Visual Situations. The Semantics of Perception Reports.* ACORD Deliverable T4.1c. FhG 1986.

[T4.2]
Hoepelman, J., Wagner, J. and Willerding, L. *Investigation of the Dialogue Manager. Literature Review.* ACORD Deliverable T4.2. FhG 1986.

[T4.3]
Hanne, K. H. and Manz, T. *Investigation of Direct Manipulation.* ACORD Deliverable T4.3. FhG 1986.

[T4.4]
Machate, J., Wagner, J. and Hoof, T. v. *Functional Specification of the Dialogue Manager.* ACORD Deliverable T4.4. FhG 1987.

[T4.7]
Manz, T. *Text Graphics Integration.* ACORD Deliverable T4.7. FhG 1988.

[T5.1]
Heyer, G., Kese, R. and Lutze, R. *Investigation on the Scope of a Knowledge Base for Business Communicaton.* ACORD Deliverable T5.1. TA 1986.

[T5.2]
Kese, R. and Lutze, R. *Evaluation of Necessary Properties of Graphical Objects.* ACORD Deliverable T5.2. TA 1986.

[T5.3]
Heyer, G., Schneider, B. and Kese, R. *Extending* PROLOG *for Processing Natural Language Semantics. PRO-LUDWIG-Manual.* ACORD Deliverable T5.3. TA 1987.

[T5.4]
Heyer, G., Luther, K. and Ovenhausen, P. *A Frame-extension of DRSs for Supporting Conceptual Reasoning.* ACORD Deliverable T5.4. TA 1988.

[T5.5]
Martin, D. *Augmentation and Modification of the Domain KB.* ACORD Deliverable

T5.5. TA 1989.

[T5.6]
Heyer, G., Luther, K., Ovenhausen, P. and Schnaterbeck, B. *Descriptions of the KB-components provided by TA.* ACORD Deliverable T5.6. TA.

[T5.7]
Luther, K., Martin, D. and Ovenhausen, P. *First implementation if the Knowledge Base Manager.* ACORD Deliverable T5.7. TA 1989.

[T5.8]
Luther, K., Martin, D. and Ovenhausen, P. *Elaboration of the KB-Manager.* ACORD Deliverable T5.8. TA 1989.

[T5.9]
Ovenhausen, P. *Final Implementation of the Knowledge Base Manager.* ACORD Deliverable T5.9. TA 1989.

[T5.10]
Heyer, G., Hoof, T. v., Luther, K. and Ovenhausen P. *Collaboration between Domain Knowledge Base and Dialog Manager.* ACORD Deliverable T5.10. TA and FhG 1988.

Publications from the Acord Project

Allgayer, J., Hinrichs, E.W., Hoepelman, J.Ph., Levelt, W., Sondheimer, N. and Wahlster W. [1987] Pointing, Language and the visual world : Towards multimodal input and output for natural language dialog systems. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, vol.2, Milan, 23-28 August 1987.

Baschung, K. [1988] "Contrôle et relations de paraphrase et d'ambiguité dans les enchâssées verbales". In *Lexique*, 6, Presses Universitaires de Lille, pp. 83-95.

Baschung, K., Bès, G.G., Corluy, A. and Guillotin, Th. [1987] Auxiliaries and Clitics in French UCG Grammar. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, Copenhague, 1-3 April 1987, pp. 173-178.

Bès, G.G. [1986] "Clíticos en francés y modelos lingüísticos". In *Revista Argentina de Lingüística*, 2, pp. 246-265.

Bès, G.G. [1988] "Clitiques et constructions topicalisées dans une grammaire GPSG du francais". In *Lexique*, 6, Presses Universitaires de Lille, pp. 55-81.

Bès, G.G. and Gardent, C. [1989] French order without order. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester Institute of Science and Technology, Manchester, UK, 10-12 April, 1989, pp. 249-255.

Bijl, A. [1985] Graphical Input: Can Computers Understand People?. In *Computers and Graphics Forum*, vol.9, no 2.

Bijl, A. [1987] Human Knowledge: AI and CAD. In *Proceedings of the 4th International Symposium on Robotics and Artificial Intelligence in Building Construction* , Israel, June, 1987.

Bijl, A. [1987] Making Drawings Talk: Pictures in Minds and Machines. In *Computers*

and Graphics Forum, vol.6, no 4, pp. 289-298.

Bijl, A., Pineda, L. and Lee, J. [1988] Notions of Language and Design for Intelligent CAD. In *Proceeding of the 2nd IFIP Workshop on Intelligent CAD, Cambridge*, North-Holland.

Bijl,A. [1988] From Graphics to Models. In *Proceedings of the EuropIA 1988*, Paris, November, 1988.

Calder, J. [1987] Typed unification for natural language processing. In Klein, E. and van Benthem, J. (eds.) *Categories, Polymorphism and Unification*, Edinburgh and Amsterdam: Centre for Cognitive Science, University of Edinburgh and Institute for Language, Logic and Information, University of Amsterdam, pp. 65-72.

Calder, J. [1989] Paradigmatic Morphology. In Moens, M. et al. (eds.) In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester Institute of Science and Technology, Manchester, UK, 10-12 April 1989, pp. 58-65.

Calder, J. and te Lindert, E. [1987] The Protolexicon: Towards a High-Level Language for Lexical Description. In Klein, E. and Benthem, J. (eds.), *Categories, Polymorphism and Unification*, Edinburgh and Amsterdam: Centre for Cognitive Science, University of Edinburgh and Institute for Language, Logic and Information, University of Amsterdam, pp. 355-370.

Calder, J., Klein, E. and Zeevat, H. [1988] Unification Categorial Grammar: A Concise, Extendable Grammar for Natural Language Processing. In *Proceedings of the 12th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Budapest, August, 1988.

Calder, J., Reape, M. and Zeevat, H. [1989] An algorithm for generation in Unification Categorial Grammar. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester, Institute of Science and Technology, Manchester, UK, 10-12 April, 1989, pp.233-240.

Dörre, J. and Momma, S. [1987] Generierung aus f-strukturen als strukturgesteuerte Ableitung. In *Proceedings of GWAI-87*, Oktober 1987, Geseke, West Germany.

Dörre, J. and Momma, S. [1987] Generation from f-structures. In Klein, E. and van Benthem, J. (eds.), *Categories, Polymorphism and Unification.* Edinburgh and Amsterdam: Centre for Cognitive Science, University of Edinburgh and Institute for Language, Logic and Information, University of Amsterdam.

Eberle, K. and Kasper, W. [1989] Tenses as Anaphora. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester, Institute of Science and Technology, Manchester, UK, 10-12 April, 1989.

Eisele, A. and Dörre, J. [1986] A Lexical Functional Grammar System in Prolog. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, August, 1986.

Eisele, A. and Dörre, J. [1988] Unification of disjunctive feature descriptions. In *Proceedings of the 26rd Annual Meeting of the Association for Computational Linguistics*, Buffalo, NY.

Gardent, C., Bès, G.G., Jurie, P.F. and Baschung, K. [1989] Efficient Parsing for French. In *Proceedings of the Conference of the 27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, BC, Canada, 26-29 June, 1989, pp. 280-287.

Heyer, G. [1988] Generic Generalizations, Discourse Representation Structures and Knowledge Representation. In Hoepelman, J. (ed.), *Representation and Reasoning, Proceedings of the Stuttgart Workshop on Discourse Representation, Dialogue Tableaux Theory*, Tübingen.

Heyer, G. [1988] A Frame Oriented Approach to Generic Descriptions. In Krifka, M. (ed.), *Genericity in Natural Language*, SNS-Bericht, Tübingen.

Heyer, G. [1989] Semantics and Knowledge Representation in the Analysis of Generic Descriptions, Journal of Semantics 7/1.

Heyer, G. and Schneider, B. [1987] Conditions on a Knowledge Representation for Processing Natural Language Semantics. In Balzert, Heyer and Lutze (eds.), *Expertensysteme 87*, pp.134-149, B.G.Teubner Verlag: Stuttgart.

Hoepelman, J.Ph. and Machate J. [1985] Dialogue Theory, theorem proving, data base questioning and natural language. In *Proceedings of the ESPRIT Technical Week 1985*, Brussels, 23-25 September 1985.

Hoof, A.J.M. v. [1988] On How to Convince a Hangman. In Hoepelman, J.Ph., (ed.), *Representation and Reasoning*, pp 47-65, M. Niemeyer Verlag: Tübingen.

Johnson, M. and Klein, E. [1986] Discourse, anaphora and parsing. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, August, 1986, pp.669-675.

Klein, E. [1986] VP ellipsis in DR theory. In Groenendijk, J., de Jongh, D. and Stokhof, M. (eds.), *Studies in Discourse Representation Theory and the Theory of Generalized Quantifiers*, pp.189-215, Dordrecht: Foris Publications.

Klein, E. [1987] DRT in Unification Categorial Grammar. In Lowden, B. G. T. (ed.), *Proceedings of Alvey Workshop on Formal Semantics in Natural Language Processing*,

March 6 1987, University of Essex, 1987.

Klein, E. [1987] Dialogues with Language, Graphics and Logic. In ESPRIT '87: Achievements and Impact: *Proceedings of the 4th Annual ESPRIT Conference*, Brussels, September 28-29, 1987, pp.867-873, Amsterdam: North Holland.

Klein, E. [1989] Grammar Frameworks. In Schnelle, H. and Bernsen, N. O. (eds.), *Logic and Linguistics*, Volume 2: Research Directions in Cognitive Science: European Perspectives, Hillsdale, N.J.: Lawrence Erlbaum Associates, pp. 71-107.

Krishnamurti, R. and Giraud, C. [1986] Towards a Shape Editor: an Implementation of a Shape Generation System. Environment and Planning B. In *Planning and Design*, vol.13, pp. 391-404.

Lee, J., Kemp, B. and Manz, T. [1989] Knowledge-based Graphical Dialogue: A Strategy and Architecture. In ESPRIT '89, Kluwer Academic Publishers.

Moens, M. and Steedman, M. [1987] Temporal Ontology in Natural Language. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, Stanford University, Stanford, Ca., 6-9 July, 1987, pp.1 -7.

Moens, M. [1988] Temporal Databases and Natural Language. In Rolland, C., Bodart, F. and Leonard, M. (eds.), *Temporal Aspects in Information Systems*, pp. 171-183. Amsterdam: North Holland.

Moens, M., Calder, J., Klein, E., Reape, M. and Zeevat, H. [1989] Expressing generalizations in unification-based grammar formalisms. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, University of Manchester Institute of Science and Technology, Manchester, UK, 10-12 April, 1989, pp. 174-181.

Momma, S. and Dörre, J. [1987] Generation from f-structures. In Klein, E. and van Benthem, J. (eds.,) *Categories, Polymorphism and Unification*. Edinburgh and Amsterdam: Centre for Cognitive Science, University of Edinburgh and Institute for Language, Logic and Information, University of Amsterdam.

Netter, K. [1986] Getting Things out of order (An LFG Proposal for the treatment of German Word Order). In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, August, 1986.

Netter, K. [1987] Wortstellung und Verbalkomplex im Deutschen. In Klenk et al. (eds), *Computerlinguistik und Philologische Datenverarbeitung*, Hildesheim.

Netter, K. [1988] Non-Local Dependencies and Infinitival Constructions in German. In Reyle, U. and Rohrer, C. (eds), *Natural Language Parsing and Linguistic Theory*,

Dordrecht: Reidel.

Netter, K. and Rohrer C. [1988] "Syntactic Analysis in Lexical Functional Grammar. The Example of German Prepositional Phrases." T.A. Informations, 1988, 28.2, Numero Spécial: Linguistique et Informatique en République Fédérale Allemande.

Plainfossé, A. [1989] ACORD Européen, In *01 Informatique Hebdo*, February, 1989.

Plainfossé, A. and Lee, J. [1989] ACORD : Construction and Interrogation of Knowledge bases using Natural Language and Graphics. In *Proceedings of HCI international 89*, Boston, pp. 837-843.

Reape, M. and Thompson, H. [1988] Parallel Intersection and Serial Composition of Finite State Transducers. In *Proceedings of the 12th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Budapest, August, 1988.

Reyle, U. and Frey, W. [1985] Grammatical Functions, Quantification and Discourse Referents. In *Proceedings of the 9th International Joint Conference on Artifical Intelligence*, 1985.

Reyle, U. [1988] Compositional Semantics for LFG. In Reyle, U. and Rohrer C. (eds.), *Natural Language Parsing and Linguistic Theories*, Reidel: Dordrecht.

Szalapaj, P. [1987] A Transformation Grammar for Line Drawings. In *Proceedings of the International Symposium on Fuzzy Systems and Knowledge Engineering*, Guangzhou-Guiyang, China, July, 1987.

Szalapaj, P. [1989] A Type-Theoretical Constructivist Semantics for Computational Systems that Depict Non-Graphical Information. In *Proceedings of the International Conference on CAD and Computer Graphics*, Beijing, China, August, 1989.

Steedman, M. [1987] Combinatory Grammars and Parasitic Gaps. In Haddock, N. J., Klein, E. and Morrill, G. (eds.,) *Edinburgh Working Papers in Cognitive Science, Volume 1: Categorial Grammar, Unification Grammar, and Parsing.* To appear in Natural Language and Linguistic Theory.

Tweed, C. and Bijl, A. [1988] MOLE: A Reasonable Logic For Design? In Hagen, P. J. W., Tomiyama, T. and Akman, V. (eds), *Intelligent CAD Systems 2: Implementation Issues*, Berlin: Springer-Verlag.

Wedekind, J. [1986] A concept of derivation for LFG. In *Proceedings of the 11th International Conference on Computational Linguistics and the 24th Annual Meeting of the Association for Computational Linguistics*, Institut für Kommunikationsforschung und Phonetik, Bonn University, Bonn, August, 1986, pp. 486-489.

Wedekind, J. [1988] Generation as Structure driven Derivation. In *Proceedings of*

the 12th International Conference on Computational Linguistics, 1988, pp. 732-737, Budapest, Hungary.

Zeevat, H. [1986] A treatment of belief sentences in Discourse Representation Theory. In Groenendijk, J., de Jongh, D. and Stokhof, M. (eds.), *Studies in Discourse Representation Theory and the Theory of Generalized Quantifiers*, pp. 189-215, Dordrecht: Foris Publications.

Zeevat, H. [1987] Text structure in Discourse Representation Theory. In Calder, J., Klein, E. and Moens, M. (eds.), *Natural Language Processing, Unification and Grammar Formalisms*, Centre for Cognitive Science, University of Edinburgh, June, 1987, pp. 59-63.

Zeevat, H., Klein, E. and Calder, J. [1987] An Introduction to Unification Categorial Grammar. In Haddock, N. J., Klein, E. and Morrill, G. (eds.,) *Edinburgh Working Papers in Cognitive Science, Volume 1:Categorial Grammar, Unification Grammar, and Parsing.*

Zeevat, H. [1988] Combining categorial grammar and unification. In Reyle, U. and Rohrer, C. (eds.), *Natural Language Parsing and Linguistic Theories*, pp. 202-229, Dordrecht: Reidel.

Zeevat, H. [1989] Realism and Definiteness. In Partee, B., Chierchia, G. and Turner, R. (eds.), *Property Theory, Type Theory and Natural Language Semantics*, Dordrecht: Reidel.

Zeevat, H. [1989] A Compositional Approach to Discourse Representation Theory. Linguistics and Philosophy, 12, pp. 95-131.

Teams

LdM **Laboratoires de Marcoussis**

Route de Nozay, 91460 Marcoussis – France

ECCS University of Edinburgh, Centre for Cognitive Science
2 Buccleuch Place, Edinburgh EH8 9LW, Scotland, UK.

EdCAAD University of Edinburgh, Department of Architecture
20 Chambers Street, Edinburgh EH1 1JZ, Scotland, UK.

CLF Université Blaise-Pascal (Clermont-Ferrand II)
Formation Doctorale Linguistique et Informatique
34, avenue Carnot, 63037 Clermont-Ferrand Cedex - France.

TA **TA Triumph-Adler**

Hundingstr. 11 bd, 8500 Nuremberg – West-Germany.

FhG Fraunhofer Institut für Arbeitswirtschaft und Organisation
Holzgartenstr. 17, 7000 Stuttgart 1 – West-Germany.

BULL **Bull-Cediag**

68 route de Versailles, 78430 Louveciennes – France.

IMS Universität Stuttgart, Institut für maschinelle Sprachverarbeitung
Keplerstr. 17, 7000 Stuttgart 1 – West-Germany.